HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Maximilian Jugl

Methoden zur markerbasierten räumlichen Vermessung auf dem mobilen Betriebssystem Android

2019

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Methoden zur markerbasierten räumlichen Vermessung auf dem mobilen Betriebssystem Android

Autor:
Maximilian Jugl

Studiengang:
Allgemeine und Digitale Forensik

Seminargruppe:
FO16w3-B

Matrikelnummer:
43606

Erstprüfer:
Prof. Dr. Thomas Haenselmann

Zweitprüfer:
M.Sc. Maik Benndorf

Mittweida, 2019

Faculty of **Applied Computer Sciences and Biosciences**

BACHELOR THESIS

Methods for marker-based spatial measuring on the mobile operating system Android

Author:
Maximilian Jugl

Course of Study:
General and Digital Forensics

Seminar Group:
FO16w3-B

Matriculation Number:
43606

First Examiner:
Prof. Dr. Thomas Haenselmann

Second Examiner:
M.Sc. Maik Benndorf

Mittweida, 2019

Bibliografische Angaben

Jugl, Maximilian: Methoden zur markerbasierten räumlichen Vermessung auf dem mobilen Betriebssystem Android, 53 Seiten, 20 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2019

Referat

In dieser Arbeit wird die Entwicklung einer Anwendung für das mobile Betriebssystem Android beschrieben, welche zwei Verfahren für die Berechnung der Entfernung zu einem kreisförmigen Marker implementiert. Hierfür werden Kernbestandteile der Auswertung von Kameradaten auf der Android-Plattform mit der OpenCV-Bibliothek erläutert und deren Einsatz in der Anwendung dargestellt. Die beiden Verfahren werden hinsichtlich ihrer Genauigkeit und Anwendbarkeit auf die mobile Plattform verglichen und ausgewertet.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

1 Motivation

Heutzutage sind Smartphones mit einer Vielzahl von Features und Sensoren ausgestattet, welche Anwendungsmöglichkeiten in einer Vielzahl an Lebensbereichen präsentieren. Mit einer weltweiten Anzahl von drei Milliarden Smartphone-Nutzern im Jahr 2018 und einer steigenden Prognose auf bis zu 3,76 Milliarden bis zum Jahr 2021 [1] haben Smartphones innerhalb des letzten Jahrzehnts eine technische Veränderung globalem Ausmaßes im Alltag veranlasst. Trotz dieser Entwicklung kam es erst in den letzten Jahren mit Fortschritten im maschinellen Lernen zu Anwendungen der räumlichen Vermessung auf der mobilen Plattform.

So hat zum Beispiel Apple mit der Veröffentlichung von iOS 12 im September 2018 eine neue Anwendung namens „Measure“ angekündigt, welche erweiterte Realität verwendet, um räumliche Vermessungen mit der Gerätekamera durchzuführen [2]. Endnutzer, die diese Anwendung verwenden möchten, müssen ein aktuelles Endgerät besitzen, um den Systemanforderungen der Anwendung zu genügen. Eine solche mit dem Betriebssystem verknüpfte Lösung existiert für die Android-Plattform noch nicht. Begründet ist dieser Sachverhalt mitunter in der breiten Verteilung der installierten und aktiven Android-Versionen. So laufen mehr als die Hälfte aller Android-Geräte auf der im August 2016 veröffentlichen Android Version 7.0 oder einer älteren Version des mobilen Betriebssystems [3]. Diese Verteilung birgt große Unterschiede in den unterstützten Features zwischen den Versionen. Unter anderem variiert die Hardwareausstattung der Endgeräte zwischen den Herstellern, was eine versions- und geräteübergreifende Lösung für die räumliche Vermessung auf der Android-Plattform erschwert.

In dieser Arbeit werden zwei Verfahren zur räumlichen Vermessung von kreisförmigen Markern vorgestellt und deren Implementierung in einer Anwendung für das mobile Betriebssystem Android beschrieben. Dabei wird auf die Integration der Kamera und der im Gerät verbauten Sensoren eingegangen. Weiterhin wird der Einsatz der Codebibliothek OpenCV im Rahmen der Anwendung dargelegt. Letztlich werden die beiden Verfahren hinsichtlich ihrer Messgenauigkeit und Anwendbarkeit auf die mobile Plattform ausgewertet.

2 Grundlagen

Wird eine Szene mit einer Kamera aufgenommen, so wird ein dreidimensionaler Raum auf eine zweidimensionale Bildebene projiziert. Dabei gehen jegliche Tiefeninformationen verloren. In den folgenden Abschnitten werden die Herausforderungen der Tiefenrekonstruktion erläutert. Methoden zur Berechnung der Entfernung zu einem kreisförmigen Marker in einem Kamerabild und die Grundlagen der Implementierung ebendieser Verfahren auf dem mobilen Betriebssystem Android werden in diesem Kapitel vorgestellt. Zusätzlich wird OpenCV als Codebibliothek für die Bildverarbeitung und für das maschinelle Sehen präsentiert.

2.1 Tiefenrekonstruktion

Die Verfahren zur Rekonstruktion von Tiefeninformationen werden in zwei Gruppen unterteilt: direkte und indirekte Verfahren. Direkte Verfahren ermitteln Tiefeninformationen durch die Zunahme zusätzlicher Signale, die zusammen mit dem Bild aufgenommen werden. Es wird also auf die Szene eingewirkt, um die Tiefe im Bild zu rekonstruieren [4, S. 104]. Beispielhaft für diese Gruppe von Verfahren sind Lichtfeldkameras. Neben dem Bild wird die Richtung der auf die Linse einfallenden Lichtstrahlen aufgenommen. Vor allem der Hersteller Lytro warb mit der Möglichkeit der nachträglichen Tiefenschärfekorrektur [5] mit dieser Methode. Da allerdings Smartphones oftmals nicht mit den notwendigen Sensoren ausgestattet sind, finden auf mobilen Plattformen eher die indirekten Verfahren Anwendung.

Indirekte Verfahren arbeiten lediglich auf der Basis eines oder mehrerer Bilder, um Tiefeninformationen zu rekonstruieren. Der bekannteste Vertreter dieser Gruppe von Verfahren ist die Stereoskopie. Wenn ein Gegenstand von zwei Orten aufgenommen wird, die zueinander sehr nah sind, dann kann die Illusion einer dreidimensionalen Aufnahme durch das Überlagern dieser Aufnahmen erzeugt werden. Umgekehrt kann durch Triangulation und dem Fokussieren eines Punktes in beiden Aufnahmen Tiefeninformationen zurückgewonnen werden [6]. Eine weitere Methode nach Saxena et al. [7] verwendet maschinelles Lernen, um aus Texturen im Bild eine Tiefenabbildung zu erstellen. Jedoch handelt es sich bei den Abbildungen nur um Schätzungen und Tiefe kann nur durch die Nähe zum Betrachter beschrieben werden. Die Korrelation zu einem Realmaß ist nicht gegeben.

In den folgenden Abschnitten werden ausgewählte Methoden zur Tiefenrekonstruktion vorgestellt. Die Algorithmen, die hinter den erläuterten Verfahren stehen, genügen den folgenden Anforderungen:

- arbeiten ausschließlich auf Bildbasis
- berechnen die Distanz in einem Realmaß
- einfach implementierbar auf mobilen Plattformen

Mit zusätzlichen Optimierungen können die vorgestellten Verfahren auch auf leistungsschwächeren Endgeräten Anwendung finden.

2.2 Methoden zur Distanzmessung

Zwei ausgewählte Methoden zur Entfernungsberechnung für den Einsatz auf mobilen Endgeräten werden auf den folgenden Seiten detailliert beschrieben. Zusätzlich werden hinführende Verfahren zur Berechnung des Sichtfeldes einer Kamera und der Korrektur von verzerrten kreisförmigen Markern erklärt.

2.2.1 Berechnung des Sichtfeldes

Für die Stereoskopie wird die Kenntnis des Sichtfeldes der Kamera vorausgesetzt. Das Sichtfeld einer Kamera setzt sich aus dem horizontalen und vertikalen Sichtwinkel zusammen — bezeichnet als ϕ_H und ϕ_V. Um das Sichtfeld zu berechnen, wird ein Gegenstand von bekannter Länge l, welcher das gesamte Sichtfeld der Kamera abdeckt, in einem definierten Abstand d parallel zum Betrachter positioniert. Der Aufbau ist in Abbildung 2.1 dargestellt.

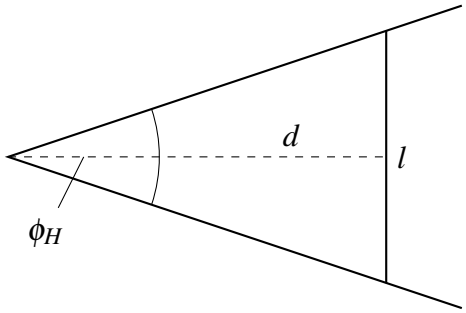

Abbildung 2.1: Berechnung des horizontalen Sichtwinkels einer Kamera

$$\tan\left(\frac{\phi_H}{2}\right) = \frac{l}{2d} \tag{2.1}$$

$$\phi_H = 2\arctan\left(\frac{l}{2d}\right) \tag{2.2}$$

Durch die gegebenen Maße lässt sich ein trigonometrischer Zusammenhang zwischen den beiden Längen und dem Sichtwinkel der Kamera herstellen [8]. Dieser Zusammen-

hang ist in Gleichung 2.1 dargestellt. Durch Umstellen der Gleichung ist es möglich, den horizontalen Sichtwinkel ϕ_H wie in Gleichung 2.2 zu berechnen. Der vertikale Sichtwinkel lässt sich analog berechnen, ist aber für weitere Betrachtungen in dieser Arbeit nicht von Relevanz.

2.2.2 Vermessung mit stereoskopischen Bildern

Ein Verfahren zur Distanzmessung auf Grundlage von stereoskopischen Bildern beschreiben Mrovlje und Vrančić [6]. Zwei identische Kameras werden parallel zueinander aufgestellt. Der zu vermessende Gegenstand muss sich dabei in der rechten Bildhälfte der linken Kamera und in der linken Bildhälfte der rechten Kamera befinden. Die beiden Kameras sind um eine Länge b voneinander entfernt. Die Entfernung zum Gegenstand kann aus dem horizontalen Versatz des Gegenstandes zwischen den beiden Kamerabildern berechnet werden. Objekte, die sich näher an den Kameras befinden, weisen einen höheren Versatz auf als ferner gelegene Objekte.

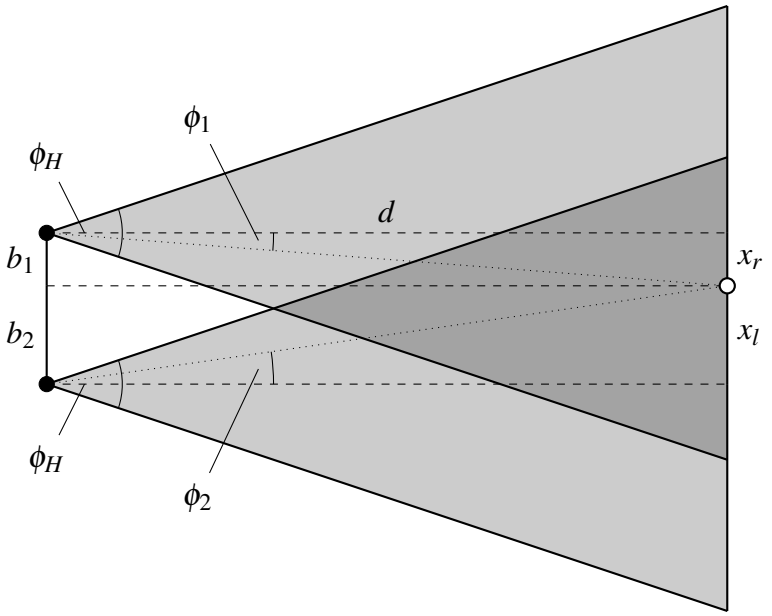

Abbildung 2.2: Distanzmessung mit stereoskopischen Bildern, basierend auf Mrovlje und Vrančić [6]

Der Messaufbau ist in Abbildung 2.2 dargestellt. Wird der Gegenstand senkrecht zur Strecke zwischen den beiden Kameras projiziert, so unterteilt sich die Strecke zwischen den Aufnahmeorten in die Längen b_1 und b_2. Zusammen mit der Entfernung zum Gegenstand bilden die Teilstrecken unter den Beobachtungswinkeln ϕ_1 und ϕ_2 der beiden Kameras rechtwinklige Dreiecke. In ihnen lassen sich trigonometrische Zusammenhänge ableiten.

$$b = b_1 + b_2 = d\,(\tan\phi_1 + \tan\phi_2) \tag{2.3}$$

$$d = \frac{b}{\tan\phi_1 + \tan\phi_2} \tag{2.4}$$

Durch Umstellen der Gleichung 2.3 lässt sich die Distanz d in Abhängigkeit der Strecke zwischen den Aufnahmeorten b und den beiden Beobachtungswinkeln ϕ_1 und ϕ_2 berechnen. Die Gleichung 2.4 beschreibt diesen Zusammenhang.

Die aufgenommenen Bilder besitzen eine feste Auflösung. Die Breite in Pixeln sei als x_0 definiert. In den Bildhälften bilden die Entfernung zum Gegenstand d und der horizontale Versatz des Gegenstandes zur Bildmitte, jeweils x_r und x_l, rechtwinklige Dreiecke. Diese können in ein Verhältnis mit der Hälfte der Bildbreite x_0 und der Hälfte des Sichtwinkels ϕ_H gesetzt werden, da sie den rechten Winkel als auch die Gegen- und Ankathete teilen.

$$\tan\phi_1 = \tan\left(\frac{\phi_H}{2}\right)\frac{2x_r}{x_0} \tag{2.5}$$

$$\tan\phi_2 = \tan\left(\frac{\phi_H}{2}\right)\frac{-2x_l}{x_0} \tag{2.6}$$

Somit können wie in Gleichung 2.5 und 2.6 die Beobachtungswinkel in Abhängigkeit vom Sichtwinkel der Kamera, der Bildbreite und dem jeweiligen horizontalen Versatz berechnet werden.

$$d = \frac{bx_0}{2\tan\left(\frac{\phi_H}{2}\right)(x_r - x_l)} \tag{2.7}$$

Durch Einsetzen der Beobachtungswinkel in die Ausgangsgleichung 2.4 ist die Berechnung der Entfernung nunmehr vom horizontalen Sichtfeld der Kamera abhängig. Somit ergibt sich die Gleichung 2.7, mit der die Entfernung zum Marker berechnet werden kann.

2.2.3 Vermessung nach Cao et al.

Die Grundlage für die markerbasierte Vermessung von Distanzen nach Cao et al. [9] ist durch das Lochkameramodell gegeben. Lichtstrahlen werden durch eine kleine Lochblende gebündelt und der abzubildende Gegenstand somit projiziert. Die Projektion eines kreisförmigen Markers mit einer Lochkamera ist in Abbildung 2.3 dargestellt.

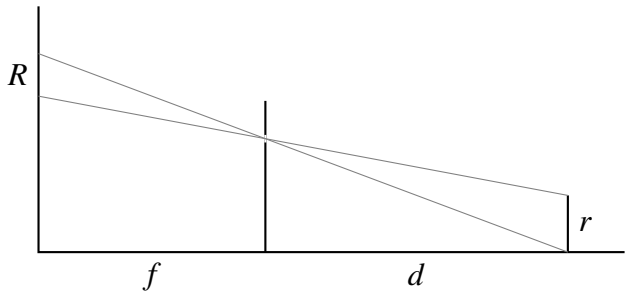

Abbildung 2.3: Projektion von kreisförmigen Markern nach dem Lochkameramodell, basierend auf Cao et al. [9]

Die Größe eines kreisförmigen Markers ist von seinem Radius r abhängig. Zwischen der Brennweite der Lochkamera f, der Gegenstandsweite d, dem Radius des Markers r und dem Radius des abgebildeten Markers R besteht somit ein festes Verhältnis.

$$\frac{f}{d} = \frac{R}{r} \tag{2.8}$$

Die Fläche eines Kreises A berechnet sich aus $A = \pi r^2$. Durch das Quadrieren beider Seiten der Gleichung 2.8 und dem Erweitern der rechten Seite um π steht die Gleichung nunmehr in Abhängigkeit von der Fläche des Markers a und der Fläche des abgebildeten Markers A.

$$\frac{f^2}{d^2} = \frac{R^2}{r^2} = \frac{\pi R^2}{\pi r^2} = \frac{A}{a} \tag{2.9}$$

Durch Umstellen der Gleichung 2.9 nach d ist es möglich, mit der Brennweite der Lochkamera und der tatsächlichen als auch der abgebildeten Fläche des Markers die Gegenstandsweite zu ermitteln.

$$d = f\sqrt{\frac{a}{A}} \tag{2.10}$$

Die Gleichung 2.10 ist von der Form des Markers unabhängig. Somit können auch Marker vermessen werden, die nicht kreisförmig sind. In dieser Arbeit werden allerdings nur kreisförmige Marker berücksichtigt. Gründe dafür sind die Einfachheit der Berechnung einer Kreisfläche und die einfache Korrektur von Verzerrungen durch die Kameraperspektive.

2.2.4 Korrektur von verzerrten Markern

Selten wird der Marker exakt kreisförmig im Kamerabild erscheinen. Stattdessen wird er häufiger als eine Ellipse abgebildet werden. Im Raum kann der Marker um seine drei Hauptachsen rotiert werden. Erkennbar wird das durch ein Stauchen und Rotieren des Markers in der Bildebene. Um die Fläche des Markers berechnen zu können, muss aus der Ellipse der Radius des abgebildeten Markers ermittelt werden.

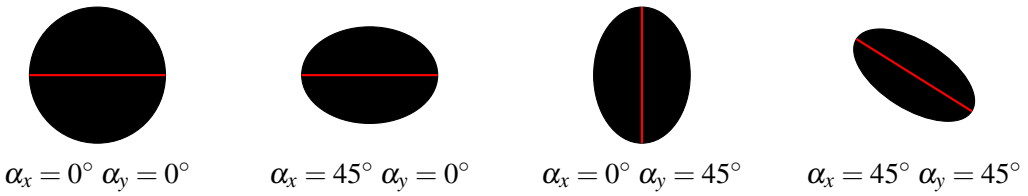

Abbildung 2.4: Position der Hauptachse von Kreismarkern nach Rotation um deren lokale x- und y-Achse

Angenommen der Ursprung des lokalen Koordinatensystems eines Markers liegt in dessen Zentrum. Die x- und y-Achse verlaufen entlang der Breite und Höhe des Markers und die z-Achse steht senkrecht auf den beiden eben genannten Achsen. Ein Marker kann um diese drei Raumachsen gedreht werden. Die z-Achse ist unwesentlich für weitere Beobachtungen, da eine Rotation um diese Achse nach der Projektion auf die Bildebene nichts an den Maßen des Markers ändert.

Nach einer Rotation um die x- oder y-Achse wird der Marker entlang der jeweils anderen Achse gestaucht. Dabei erscheint der Marker in der Bildebene als eine Ellipse, dessen Hauptachse dem Durchmesser des Markers entspricht [9]. Auch bei einer kombinierten Rotation um die x- als auch der y-Achse bleibt die Länge der Hauptachse erhalten. Der Radius eines verzerrten Markers kann somit aus der Hälfte der Länge der Hauptachse seiner Ellipsenform ermittelt werden.

2.3 Android und OpenCV

Die im vorherigen Abschnitt beschriebenen Methoden werden in einer Android-Anwendung implementiert. Zentraler Bestandteil jeder Anwendung auf Android sind sogenannte Activities, welche den Ablauf einer Applikation kontrollieren. Neben einer Begriffsklärung zur Activity wird in diesem Abschnitt der Activity-Lebenszyklus in Android vorgestellt. Folglich werden die Kamera und die Sensor API des mobilen Betriebssystems näher erläutert. Weiterhin wird die Integration der OpenCV-Bibliothek in die Android-Plattform beschrieben und ihre Kernbestandteile, welche Anwendung in dieser Arbeit finden, beschrieben.

2.3.1 Activity-Lebenszyklus

In der Android-Plattform beschreibt eine Activity eine interaktive Nutzeroberfläche mit eigener Anwendungslogik [10]. Der Nutzer navigiert innerhalb einer Anwendung oftmals durch mehrere Activities, welche unterschiedliche Aufgaben erfüllen. Jede Anwendung besitzt mindestens eine Haupt-Activity, welche beim Start der Applikation durch den Nutzer oder durch das Betriebssystem aufgerufen wird.

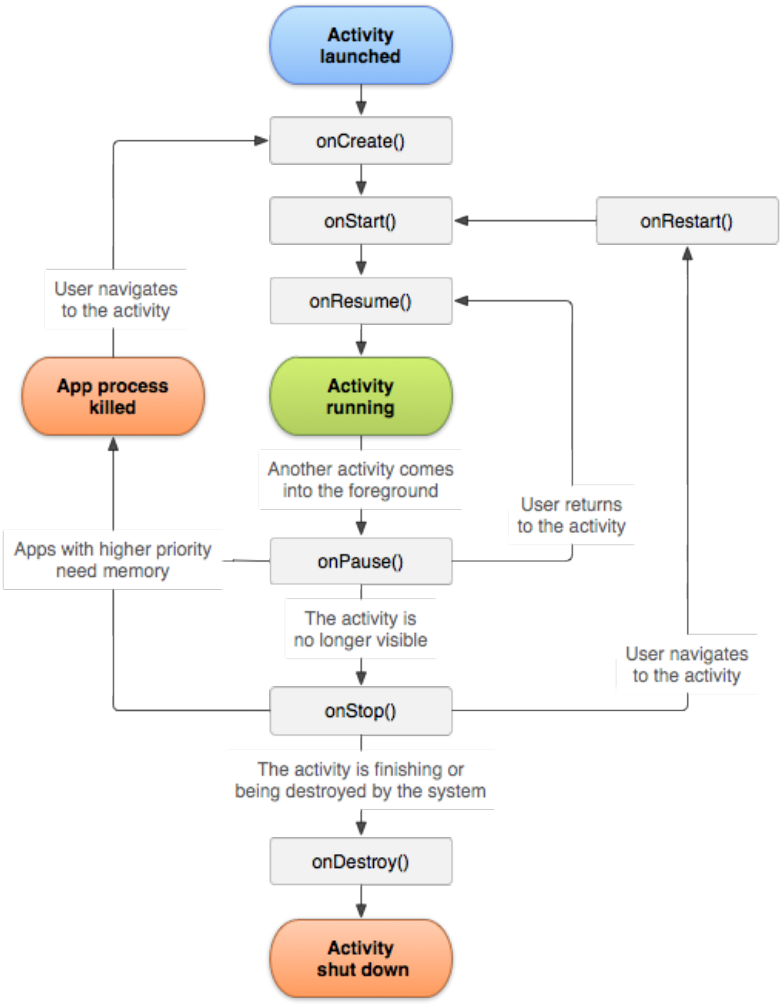

Abbildung 2.5: Ablaufdiagramm des Activity-Lebenszyklus, übernommen von Android API [11]

Das Schema des Lebenszyklus einer Activity ist in Abbildung 2.5 dargestellt. Jede Activity besitzt Methoden, welche die Activity über Änderungen am eigenen Status durch das Betriebssystem informiert [11]. Diese Methoden können überschrieben werden, um das Verhalten der Activity zur Laufzeit zu beeinflussen. Die sechs Hauptmethoden und ihre Funktion werden in den folgenden Absätzen erläutert.

Die Funktion `onCreate` wird nach dem Erstellen der Activity durch das Betriebssystem aufgerufen. Wenn die Activity an eine Benutzeroberfläche gebunden ist, so muss sie hier festgelegt und initialisiert werden. Bevor die Activity für den Benutzer sichtbar ist,

wird die Funktion `onStart` aufgerufen. In ihr werden üblicherweise Operationen an den Bildschirmkomponenten vorgenommen, sodass diese interaktiv werden. Sobald die Activity in den Fokus des Nutzers rückt, wird die Funktion `onResume` aufgerufen. In diesem Zustand verweilt die Activity, bis der Nutzer zu einer anderen Activity navigiert.

Sobald die Funktion den Fokus verliert, wird `onPause` aufgerufen. Wenn bestimmte Operationen nur ausgeführt werden sollen, wenn die Activity im Fokus ist, so müssen sie im Zuge des Funktionsaufrufs pausiert werden. Erst wenn die Activity nicht mehr sichtbar ist, wird `onStop` aufgerufen. In ihr sollten Anwendungsdaten und der aktuelle Status der Benutzeroberfläche gespeichert werden, sofern der Nutzer zurück zur Activity navigiert und diese Daten wiederhergestellt werden müssen. Wird die Anwendung geschlossen oder durch das Betriebssystem beendet, so wird `onDestroy` aufgerufen. Anwendungsressourcen, die noch nicht durch andere Funktionsaufrufe im Lebenszyklus freigegeben wurden, sollten in diesem Schritt bearbeitet werden. Die Kenntnis des Activity-Lebenszyklus ist für den Aufbau der Anwendung im Rahmen dieser Arbeit von großer Bedeutung.

2.3.2 Kamera API

Die Android Kamera API erlaubt den Zugriff auf eine oder mehrere Gerätekameras zum Aufnehmen einzelner Bilder, Bildfolgen und dem Präsentieren einer Kameravorschau. Sämtliche Kameraparameter können vom Entwickler kontrolliert werden [12]. Für diese Zwecke muss vorher eine Berechtigung zum Aufnehmen und Verarbeiten von Kamerabildern vom Nutzer eingefordert werden. Die Kamera API wurde mit der Einführung von Android 5 im Oktober 2015 erneuert. Jedoch wurden die nunmehr veralteten Klassen nicht ersetzt, sondern lediglich in der Dokumentation der Android API als obsolet markiert. Stattdessen soll die aktuellere `camera2` API verwendet werden. Zusätzlich zu Verbesserungen in ihrer Leistung erlaubt sie eine feinere Kontrolle über Kameraparameter als ihr Vorgänger.

Der Ablauf zum Erstellen einer Kameravorschau hat sich zwischen den beiden APIs nicht einheitlich geändert. Zuerst muss der Zugriff auf eine Gerätekamera erlangt werden. In der veralteten `camera` API wurde der Zugriff über die Klasse `Camera` geregelt. Der Zugriff über die `camera2` API erfolgt dahingegen über die Klasse `CameraManager`. Die Ausgabe der Kamera wird auf eine `SurfaceView` gezeichnet. Die `SurfaceView` ist eine Bildschirmkomponente, welche sich wie eine Zeichenoberfläche verhält. Sie findet Anwendung in Spielen als auch zum Anzeigen von Kamerabildern. Der Zugriff auf rohe Pixeldaten des Kamerabildes ist auch möglich. Eine `SurfaceView` ist immer zugleich ein Inhaber eines `SurfaceHolder`s, der das Format und die Parameter des Ausgabebildes kontrollieren kann. Zuletzt wird eine sich selbst wiederholende Anfrage an die Kamera gesendet. Ihr Rückgabewert ist stets das aktuelle Kamerabild. Somit können Bilder von der Kamera als Teil der interaktiven Benutzeroberfläche angezeigt werden.

Unter anderem stellt Android die experimentelle Bibliothek CameraX bereit [13]. Ihr Ziel ist die Vereinfachung der Verwendung der `camera2` API. Sie übernimmt die Verbindung zur Kamera und deren Konfiguration. Der Entwickler muss nur spezifizieren, welche Operationen auf den Kamerabildern ausgeführt werden sollen. Die Bibliothek befindet sich noch im Alpha-Stadium weswegen sie in dieser Arbeit keine Anwendung findet. In Zukunft ist es jedoch abzusehen, dass die Bildanalyse auf Android für Entwickler somit vereinfacht wird.

2.3.3 Sensor API

Um Messwerte aus der Umwelt zu erhalten, ermöglicht Android den Zugriff auf die im Gerät verbauten Sensoren. Die Dokumentation listet bis zu 13 verschiedene Sensoren, die je nach Hardware- und Softwareausstattung abgefragt werden können [14]. Im Rahmen der Stereoskopie muss sichergestellt werden, dass sich die Ausrichtung des Gerätes zwischen den Aufnahmeorten nicht oder nur geringfügig ändert. Android stellt dafür zwei zusammengesetzte Sensoren zur Verfügung, deren Anwendbarkeit in den folgenden Absätzen diskutiert wird.

Ausrichtungssensor

Die Messwerte des Ausrichtungssensors ergeben sich aus einer Kombination des im Gerät verbauten Beschleunigungsmessers und des Magnetfeldstärkenmessers. Die Lage wird als Rotation im Raum um die drei Hauptachsen im Bogenmaß beschrieben. Die Achsen sind relativ zur natürlichen Ausrichtung des Bildschirms und sind in Abbildung 2.6 dargestellt.

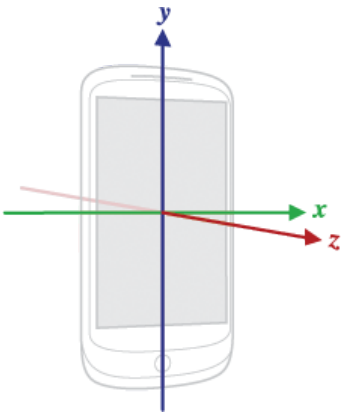

Abbildung 2.6: Koordinatensystem der Android Sensor API, übernommen von Android API [15]

An dem starken Rauschen der beiden zugrunde liegenden Sensoren leidet die Genauigkeit des Ausrichtungssensors. Seit der Veröffentlichung von Android 4 ist der zusammengesetzte Sensor als veraltet gekennzeichnet. Stattdessen sollen Entwickler die Sensoren, aus denen sich der Ausrichtungssensor zusammensetzt, direkt abfragen. Für

das Berechnen der Geräteausrichtung aus den gemeldeten Werten der beiden Sensoren stellt Android die Funktion `getOrientation` der Klasse `SensorManager` bereit [16]. Das Rauschen der Messwerte wird dadurch allerdings nicht minimiert.

Drehvektorsensor

Zusätzlich zum Beschleunigungsmesser und Magnetfeldstärkenmesser verwendet der Drehvektorsensor die Messwerte des im Gerät verbauten Gyroskops. Das Gyroskop ermöglicht eine weitaus genauere Lagebeschreibung des Gerätes im Raum. Weiterhin wird die Lage des Gerätes nicht als Rotation um dessen Hauptachsen beschrieben, sondern als Einheitsquaternion.

$$q = w + x\mathrm{i} + y\mathrm{j} + z\mathrm{k} \tag{2.11}$$

Quaternione sind als vierstelliges Zahlentupel definiert. Sie bestehen wie in Gleichung 2.11 dargestellt aus einem Realteil w und einem Imaginärteil, der sich aus den drei Komponenten x, y und z zusammensetzt. Die beiden Teile werden jeweils auch als Skalarteil und Vektorteil bezeichnet. Eine Einheitsquaternion, wie sie in der Lagebeschreibung auf Android verwendet wird, besitzt einen Betrag von eins. Quaternione ermöglichen die elegante Beschreibung von Ausrichtungen im Raum und die einfache Berechnung von Vektorrotationen [17, S. 46].

Android ermöglicht die Umrechnung von Quaternionen in eulersche Winkel. Aufgrund der höheren Genauigkeit und der geringeren Anfälligkeit gegenüber Rauschen in den Messdaten, ist der Drehvektorsensor dem Ausrichtungssensor zu bevorzugen.

2.3.4 OpenCV

OpenCV ist eine quelloffene Bibliothek mit einer Vielzahl an Algorithmen im Bereich des maschinellen Sehens [18]. Der Quellcode ist in C++ geschrieben und unter der BSD-Lizenz verfügbar. Portierungen für die Programmiersprachen Python und Java sind ebenfalls Kernbestandteil der Weiterentwicklung der Bibliothek. Letztere hat zur Folge, dass auch eine Variante von OpenCV für die Android-Plattform angeboten wird. In den folgenden Absätzen wird der grundlegende Aufbau von OpenCV für Android beschrieben und auf Besonderheiten im Umgang mit der Bibliothek eingegangen. Die Anwendung, die im Rahmen dieser Arbeit entwickelt wurde, verwendet OpenCV in der Version 4.1.0.

Kamerabilder mit der JavaCameraView

Der Zugriff auf Kernmodule der OpenCV-Bibliothek wird durch native Funktionsaufrufe ermöglicht. Auf Android wird der Quellcode um Hilfsklassen und Bildschirmkomponenten erweitert, die den Umgang mit OpenCV auf der Android-Plattform erleichtern. Vor allem die Komponenten `JavaCameraView` und `JavaCamera2View` sind von Interesse. Beide zeigen ein Vorschaubild der Kamera an und stellen aufgenommene Bilder für die Verwendung mit der OpenCV-Bibliothek bereit. Der Unterschied zwischen den beiden Komponenten liegt in den verwendeten Kamera API-Versionen. Erstere verwendet die veraltete `camera` API und Letztere bedient sich der aktuellen `camera2` API.

Kamerabilder werden im `Mat`-Datentyp bereitgestellt. Der `Mat`-Datentyp repräsentiert eine Matrix mit $n \times m$ Zellen. Jede `Mat` legt einen Datentyp für die Werte innerhalb dieser Zellen fest. Weiterhin kann eine `Mat` mehrere Kanäle besitzen in Anlehnung an Farbkanäle in einem Kamerabild. Aufgenommene RGB-Bilder auf Android werden in OpenCV dementsprechend durch Matrizen mit drei Kanälen dargestellt, deren Spalten- und Zeilenanzahl der Breite und Höhe des Bildes in Pixeln entsprechen. Da ein Großteil aller OpenCV-Funktionen auf dem `Mat`-Datentyp arbeiten, ist somit eine nahtlose Verarbeitung von Kamerabildern auf Android möglich.

Die Implementierung der Verbindung von Systemkamera und OpenCV weist erhebliche Leistungsunterschiede zwischen der `camera` API und der `camera2` API auf [19]. Die CPU-Auslastung in Verbindung mit der `camera2` API ist wesentlich höher, die Bildrate ist um mindestens 50 % geringer und auf einigen Geräten ist die Auswahl der Bildformate unvollständig. Aus diesem Grund werden in dieser Arbeit auf Funktionen und Klassen der veralteten `camera` API verwiesen. Da diese API zu jedem Zeitpunkt entfernt werden kann, sollte sich auf ihre Funktionstüchtigkeit auf jedem Endgerät nicht verlassen werden.

Statische und dynamische Initialisierung

Vor der Verwendung von OpenCV in Android-Anwendungen muss die Codebibliothek initialisiert werden. Die Initialisierung kann auf zwei Arten durchgeführt werden: asynchron oder statisch. Bei der asynchronen Initialisierung wird vorausgesetzt, dass der Nutzer die Anwendung „OpenCV Manager“ installiert hat. Statt den gesamten kompilierten Quellcode der OpenCV-Bibliothek in die Anwendung zu packen, realisiert der „OpenCV Manager“ eine Art dynamische Verlinkung der Bibliothek zur Laufzeit. Diese Anwendung ist seit der Veröffentlichung von OpenCV 3.0.0 nicht mehr in Google's Play Store zum Download verfügbar [20]. Unter anderem wird mit OpenCV 4.0.0 der „OpenCV Manager“ nicht mehr als gepackte Anwendung zum Installieren mittels der Android Debug Bridge von der OpenCV-Seite angeboten. Die asynchrone Initialisierung hat somit mit der aktuellen Version von OpenCV keine Anwendung.

Bei der statischen Initialisierung wird vorausgesetzt, dass der Quellcode der OpenCV-Bibliothek mit der Anwendung gepackt wird. Die OpenCV-Funktionen und Module werden direkt aus der Anwendung geladen. Die Größe der Anwendung steigt dadurch auf ein Vielfaches an. In der Dokumentation zur OpenCV-Version 2.4.13.7 für Android wird die statische Initialisierung als veraltet bezeichnet [21]. Da für die in dieser Arbeit verwendeten Version von OpenCV allerdings keine andere Methode zur Verfügung steht, wird in der Anwendung auf die statische Initialisierung zurückgegriffen.

3 Umsetzung

Die Konzepte aus dem vorhergehenden Kapitel werden in der Android-Anwendung „AndMeasure“ implementiert und umgesetzt. In den folgenden Abschnitten wird die Entwicklung der Anwendung geschildert. Zuerst werden die grundlegenden Bestandteile der Anwendung beschrieben. Daraufhin wird die Implementierung der Messverfahren erläutert. Grundlagen der Android-Programmierung werden für dieses Kapitel vorausgesetzt.

3.1 Grundlegender Aufbau der Anwendung

Die Implementierung der Messverfahren stützt sich auf einen großen Quellcodeanteil, der in der gesamten Anwendung häufig wiederverwendet wird. In diesem Abschnitt werden der Aufbau der Benutzeroberfläche und die grundlegende Struktur der Anwendung dargestellt. Die Integration der Android-Plattform und der OpenCV-Bibliothek in die Anwendung wird beschrieben und einzelne Hilfsklassen werden hervorgehoben.

3.1.1 Paketstruktur und Benutzeroberfläche

Die Anwendung ist in drei Pakete untergliedert: `activity`, `fragment` und `util`. Das Paket `activity` enthält alle Unterklassen von `Activity`. Analog enthält das Paket `fragment` alle Unterklassen von `Fragment`. Ein Fragment ist ein Teil der Benutzeroberfläche mit eigener, von der Activity unabhängiger, Anwendungslogik. Einzelne Bestandteile einer Anwendung können somit modularisiert und wiederverwendet werden [22]. Üblicherweise besitzt eine Activity ein oder mehrere Fragments. Das Paket `util` enthält Hilfsklassen, die anderweitig nicht in die anderen beiden Pakete eingeordnet werden können und in der gesamten Anwendung verwendet werden.

Der Startpunkt der Anwendung ist die Klasse `MainActivity`. Ihre Aufgabe besteht in der Organisation der Nutzereingaben. Für jedes implementierte Messverfahren sind die Eingabemasken in ihren eigenen Fragments gruppiert. Diese werden, wie in Abbildung 3.1 dargestellt, in einem `ViewPager` zusammengefasst. Dabei handelt es sich um eine Bildschirmkomponente, welche die ihr zugeordneten Fragments in Reitern zur einfachen Navigation durch den Nutzer organisiert.

Jedes Messverfahren ist in der Anwendung so implementiert, dass es ein Fragment für die Nutzereingaben und eine Activity für die Durchführung der Messung besitzt. Eine Activity wird nach der Eingabe der Parameter für die Messung durch das zugeordnete Fragment gestartet. Das Fragment übergibt dabei die Nutzereingaben als Parameter an die Activity. Nach der erfolgreich abgeschlossenen Messung gibt die Activity die

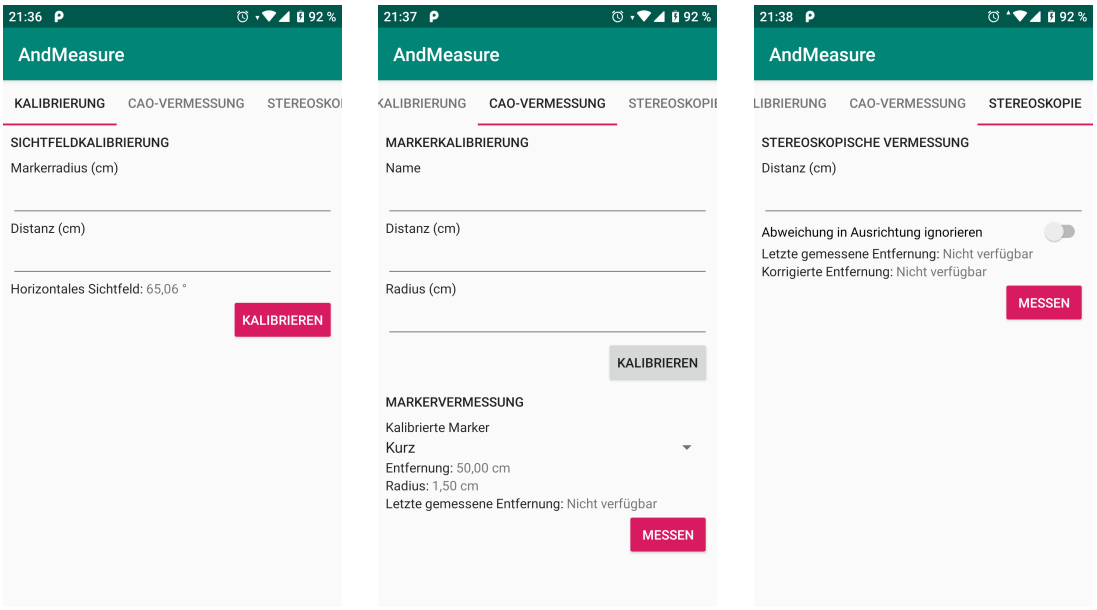

Abbildung 3.1: Organisation der Nutzereingaben in der MainActivity

Messergebnisse an das Fragment zurück. Dort werden sie je nach Verfahren angezeigt, gespeichert oder weiterverarbeitet.

3.1.2 Die Klasse BaseCvCameraActivity

Für die Gestaltung einer interaktiven Messung ist jedes der zuvor genannten Verfahren in seiner eigenen Activity implementiert. Beim Start einer Messung muss die Kameravorschau vorbereitet werden und die Benutzeroberfläche dementsprechend angepasst werden. Nach dem Abschluss einer Messung muss die Anwendung wieder in ihren Ursprungszustand zurückversetzt werden. Um Wiederholungen des gleichen Quellcodes in jedem implementierten Messverfahren zu vermeiden, sind die Start- und Stoppoperationen in der Klasse `BaseCvCameraActivity` realisiert. In den folgenden Absätzen wird ihre Arbeitsweise näher erläutert.

Integration in den Activity-Lebenszyklus

Die Klasse `BaseCvCameraActivity` überschreibt einige Methoden des Activity-Lebenszyklus um Start- und Stoppoperationen, wie in Abbildung 3.2 dargestellt, durchzuführen. Beim Start der Activity wird die Funktion `onCreate` aufgerufen. Die Anwendung wechselt in den Vollbildmodus. Dafür wird die Benutzeroberfläche in das Hochformat versetzt, die Statusleiste des Android-Systems versteckt und ein Flag gesetzt, welches das Display zwingt während der Laufzeit der Activity den Fokus zu behalten. Die Layout-Datei für die Benutzeroberfläche, welche mit der `BaseCvCameraActivity` verbunden ist, enthält als einzige Komponente eine `JavaCameraView`. Ihr wird mitgeteilt, dass sie Kamerabilder, sobald sie verfügbar sind, an die Activity übergeben soll.

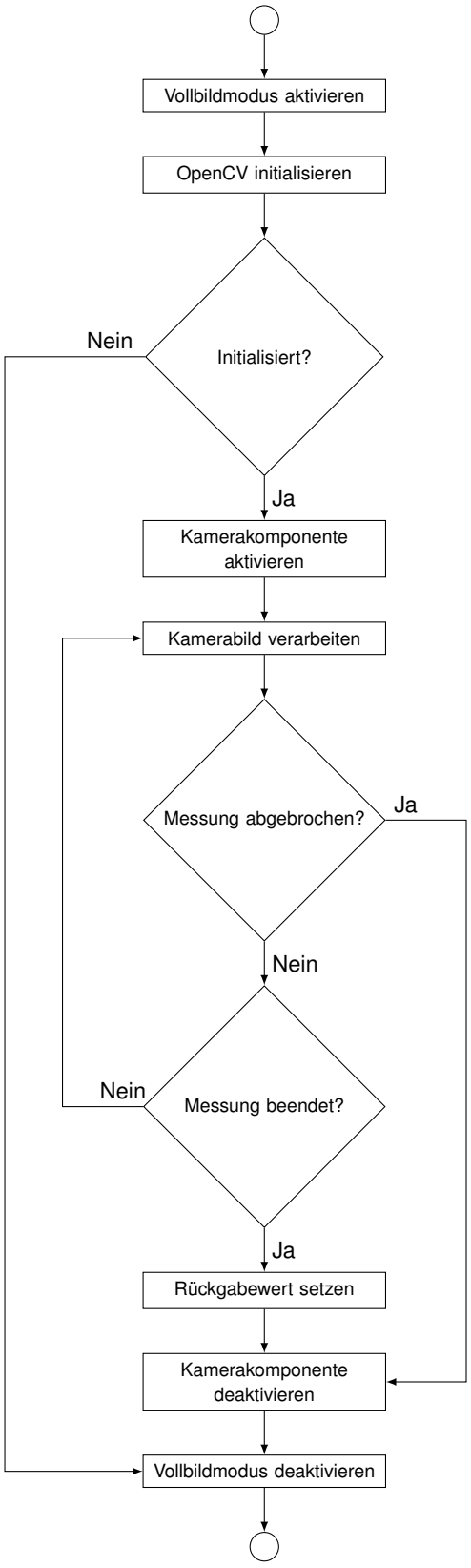

Abbildung 3.2: Ablaufdiagramm der Klasse BaseCvCameraActivity

In der Funktion `onResume` wird die statische Initialisierung der OpenCV-Bibliothek durchgeführt. Wurde OpenCV erfolgreich initialisiert, so wird die Kamerakomponente aktiviert. Erst ab diesem Punkt können OpenCV-Ressourcen verwendet werden. Scheitert die Initialisierung von OpenCV, so wird die Activity sofort beendet.

Der Nutzer kann nun mit der Activity interagieren. Jedes implementierte Messverfahren führt an dieser Stelle Messungen und Berechnungen durch. Der Nutzer oder die Activity selbst können die Activity beenden um die Messung abzuschließen oder abzubrechen. Intern wird daraufhin die Funktion `onPause` aufgerufen. In ihr wird die Kamerakomponente deaktiviert. Mit dem Aufruf der Funktion `onDestroy` wird der Vollbildmodus beendet und das in `onCreate` gesetzte Flag gelöscht.

Integration in den OpenCV-Prozess

Die Activity erbt von der OpenCV-Schnittstelle `CvCameraViewListener2`. Damit erhält sie drei Methoden, welche die Activity über den Status der mit ihr verknüpften Kamerakomponente informiert. Diese Methoden werden in folgender Reihenfolge aufgerufen:

- `onCameraViewStarted` nachdem die Kamerakomponente aktiviert und die Verbindung zur Gerätekamera erfolgreich hergestellt wurde
- `onCameraFrame` sobald ein neues Kamerabild verfügbar ist
- `onCameraViewDestroyed` nachdem die Kamerakomponente deaktiviert wurde

Die Funktion `onCameraViewStarted` wird zusammen mit der Breite und Höhe der Kamerabilder aufgerufen. Somit können OpenCV-Ressourcen auf Grundlage der Maße der Kameravorschau allokiert werden. Ebendiese Ressourcen müssen mit dem Aufruf der Funktion `onCameraViewDestroyed` freigegeben werden, um Speicherlecks zu vermeiden.

Kamerabilder werden mit der Funktion `onCameraFrame` als eine Instanz der OpenCV-Schnittstelle `CvCameraViewFrame` an die Activity übergeben. Sie ermöglicht das Umwandeln eines Kamerabildes in eine RGBA-Matrix oder eine Graustufenmatrix. Diese können in der Anwendung zu Analyse- oder Verarbeitungszwecken verwendet werden. Weiterhin wird die von der Funktion `onCameraFrame` zurückgegebene Matrix in die Benutzeroberfläche gezeichnet. Somit ist es möglich zu kontrollieren, was dem Nutzer in der Kameravorschau angezeigt wird. Standardmäßig gibt die Activity die RGBA-Matrix ohne Veränderungen zurück.

Interaktion mit dem Nutzer

Dem Nutzer soll es möglich sein, Regionen in das Kamerabild einzuzeichnen damit auf Grundlage des ausgewählten Bereiches Berechnungen und Messungen durchgeführt

werden können. Activities können die Funktion `onTouchEvent` überschreiben, um über Berührungsereignisse informiert zu werden. Die Koordinaten eines Berührungsereignisses beziehen sich auf die physischen Pixel, die im Bildschirm des Gerätes verbaut sind. Aufgrund von technischen Beschränkungen kann es passieren, dass die Kameravorschau eine geringere Auflösung als der Bildschirm besitzt. Das hat zur Folge, dass Bildschirmkoordinaten nicht den Matrixkoordinaten eines Bildes entsprechen.

Die Klasse `BaseCvCameraActivity` behandelt dieses Problem, indem sie die Funktion `onTouchEvent` überschreibt und in ihr die Bildschirmkoordinaten in Matrixkoordinaten umrechnet. Weiterhin definiert sie die Schnittstelle `CvMatTouchListener`. Sie enthält Funktionen, welche zusammen mit den Matrixkoordinaten aufgerufen werden, wenn der Nutzer seinen Finger absetzt, bewegt und anhebt. Klassen, die über Berührungsereignisse relativ zur Bildmatrix informiert werden wollen, implementieren diese Schnittstelle und registrieren sich als Listener in der Activity. Somit ist es möglich auf Berührungsereignisse zu reagieren und diese in der Kameravorschau darzustellen.

3.1.3 Fixierter Kamerafokus

Die im vorherigen Abschnitt erwähnte `JavaCameraView` durchläuft im Hintergrund mehrere Schritte, um Bilder von der Gerätekamera zu erhalten. Zuerst muss die Kamera an der Rückseite des Gerätes selektiert werden. Da aktuellere Smartphonemodelle meist über mehrere Kameras verfügen, muss zusätzlich die beste Kamera für den Anwendungsfall der Bildanalyse anhand ihrer Eigenschaften ausgewählt werden. Als Nächstes muss ein Aufnahmemodus festgelegt werden. Dieser bestimmt ob nur ein oder mehrere Bilder an die Anwendung übergeben werden sollen. Der Modus beeinflusst aber auch Kameraparameter, wie die Blende und den Fokus, welche automatisch von Android geregelt werden. Nach der Festlegung des Modus können weiterhin einzelne Kameraparameter geändert werden, allerdings legen in den meisten Fällen die Modi eine solide Grundlage, sodass nachträgliche Änderungen nicht erforderlich sind.

Im Vorschaumodus, welcher von OpenCV standardmäßig angefordert wird, hat die Bildrate eine höhere Priorität als die Bildqualität. Das Bild wird nicht vorverarbeitet und die Kameraparameter werden so eingestellt, dass das Bild immer einen bestimmten Gegenstand fokussiert. Dieser Softwarefokus sucht stets einen Gegenstand, welcher im Interesse des Nutzers sein könnte. Das ist für Videoaufnahmen und Szenen ohne große Tiefenunterschiede wünschenswert, jedoch nicht für räumliche Vermessungen. Idealerweise sollte stets der Marker im Fokus der Kamera sein, jedoch kann es mit dem in OpenCV voreingestellten Modus nicht gewährleistet werden. Die Fixierung des Kamerafokus ist wichtig, um konsistente Messergebnisse zu garantieren.

Da OpenCV das Einstellen von Kameraparametern in der in dieser Arbeit verwendeten Version nicht unterstützt, muss der Quellcode in der Bibliothek selbst geändert werden.

```
170 List<String> FocusModes = params.getSupportedFocusModes();
171 if (FocusModes != null && FocusModes.contains(Camera.Parameters.FOCUS_MODE_INFINITY
        ))
172 {
173     params.setFocusMode(Camera.Parameters.FOCUS_MODE_INFINITY);
174 }
```

Listing 3.1: Geänderte Fokuseinstellung in der Klasse JavaCameraView

Das Listing 3.1[1] zeigt die geänderte Stelle in der Codebibliothek. An dieser Stelle würde normalerweise der Fokusparameter auf `FOCUS_MODE_CONTINUOUS_VIDEO` gesetzt werden. Stattdessen wird er auf `FOCUS_MODE_INFINITY` festgelegt. Dieser Modus ist speziell dafür vorgesehen, den Fokus der Kamera auf ein Objekt zu fokussieren, welches sich theoretisch unendlich weit von der Kamera entfernt befindet. Somit befindet sich der Großteil einer aufgenommenen Szene im Fokus der Kamera.

3.1.4 Die Klasse MatProcessor

Bevor Berechnungen für einen Marker durchgeführt werden können, muss dieser im Kamerabild lokalisiert werden. OpenCV stellt hierfür Kamerabilder als Matrizen zur Verwendung mit den OpenCV-Funktionen bereit. In den folgenden Absätzen wird erläutert, wie ein Kamerabild mit der Klasse `MatProcessor` binarisiert wird. Weiterhin wird die Suche nach Konturen im vorverarbeiteten Kamerabild beschrieben.

Einstellung der Parameter

Die Bildvorverarbeitung und Kontursuche sind von einer Vielzahl an Parametern abhängig, die je nach Messverfahren oder Umwelteinflüssen angepasst werden müssen. Einstellbare Parameter sind in einer `Map` von `String`s auf `Object`s gespeichert. Die Schlüsselwerte sind als statische Felder mit dem Präfix `PARAM` verfügbar. Um einen Parameter zu ändern, muss die Funktion `set` aufgerufen werden. Ihre Argumente sind der Schlüssel des zu ändernden Parameters und dessen neuer Wert. Der aktuelle Wert eines Parameters kann mit der Funktion `get` ermittelt werden. Jeder Parameter hat einen Standardwert, der bei der Erzeugung einer neuen Instanz der Klasse `MatProcessor` festgelegt wird. Die Standardwerte sind in Tabelle 3.1 aufgeführt. Während der Beschreibung der Bildvorverarbeitung und Kontursuche wird der Einfluss der Parameter und die Wahl der Standardwerte erläutert.

[1] Die Zeilennummern neben den Listings in dieser Arbeit beziehen sich auf die Zeilen in den Quellcodedateien.

Tabelle 3.1: Standardwerte der Parameter in der Klasse MatProcessor

Parameter	Datentyp	Standardwert
BILATERAL_SIGMA	float	50
BLUR_FILTER	FilterType	GAUSSIAN
BLUR_KERNEL_LENGTH	int	5
CIRCLE_ASPECT_THRESHOLD	float	0,1
MIN_CONTOUR_AREA	int	50
THRESH_MAXVAL	int	255

Vorverarbeitung

Die Funktion `preprocess` führt Vorverarbeitungsschritte auf einer Graustufenmatrix durch, um ein binarisiertes Bild wie in Abbildung 3.3 zu erzeugen. Im ersten Vorverarbeitungsschritt wird ein Weichzeichner auf die Matrix angewandt, um das Rauschen im Bild durch Datenkompression und Hintergrundobjekte zu reduzieren. Die Art des Weichzeichners wird durch den Parameter `BLUR_FILTER` beeinflusst. Implementiert sind der Boxfilter, der Gaußsche Weichzeichner und der bilaterale Filter. Die Seitenlänge des Weichzeichenkernels ist durch den Parameter `BLUR_KERNEL_LENGTH` festgelegt.

Abbildung 3.3: Binarisierung eines Kamerabildes mit der Klasse MatProcessor. Links: Originalbild. Rechts: Bild nach Vorverarbeitung.

Die Standardwerte für die beiden Parameter ergeben sich aus einer Reihe von Tests zum Vergleich der Leistung der drei Filter. Der Boxfilter ist schnell, behält aber Kanten im Bild schlecht bei. Dahingegen zeichnet sich der bilaterale Filter mit dem Erhalt von Objektkanten aus, ist aber sehr rechenintensiv. Der Gaußsche Weichzeichner bietet den besten Kompromiss aus dem Erhalt von Kanten und benötigter Rechenzeit. Für die Seitenlänge des Kernels wurde die Kernelgröße inkrementell erhöht und auf das Kamerabild angewandt. Der Rechenaufwand eines Weichzeichners steigt mit der Größe des Kernels. Die Seitenlänge wurde erhöht, bis die Bildrate durch den Weichzeichner merklich vermindert wurde. Somit ergibt sich der Gaußsche Weichzeichner als Standardfilter und die Seitenlänge von 5 px als Standardwert für den Weichzeichenkernel. Diese Werte hängen von der Rechenleistung des Endgerätes ab. Sie sind für das Testgerät, welches während der Entwicklung verwendet wurde, optimal.

Zusätzlich kann für den bilateralen Filter der Parameter `BILATERAL_SIGMA` festgelegt werden. Er beeinflusst, wie groß der Farbunterschied zweier Pixel sein darf, damit die-

se zusammengemischt werden. Der Standardwert von 50 entstammt der OpenCV-Dokumentation zur Funktion `bilateralFilter` als Mittelwert zwischen schwachem und starkem Zusammenmischen von Farben [23].

Nach dem Weichzeichnen wird das Bild normiert. Dabei werden die Grauwerte aufgespreizt, sodass der gesamte Wertebereich von 0 bis 255 ausgenutzt wird. Im letzten Schritt werden die Grauwerte mittels Schwellwertbildung im Bild binarisiert. Der Grenzwert für Grauwerte wird durch den in OpenCV implementierten Otsu-Algorithmus bestimmt. Dieser wählt den Grenzwert auf Grundlage der Häufigkeitsverteilung der Grauwerte im Bild. Alle Grauwerte, die über diesem Grenzwert liegen, werden auf den Wert des Parameters `THRESH_MAXVAL` gesetzt. Das Ergebnis ist ein binarisiertes Bild, in dem dunkle Bereiche schwarz und helle Bereiche weiß sind.

Kontursuche

Auf einem binarisierten Bild kann eine Kontursuche durchgeführt werden. OpenCV stellt hierfür die Funktion `findContours` bereit [24]. Sie sucht in einem binarisierten Bild nach Konturen und gibt diese zurück. Konturen werden als Liste von Punkten, welche die Kontur approximieren, gespeichert [25, S. 12]. Die Funktion `findContours` ist komplex und akzeptiert eine Vielzahl von Parametern. Deswegen implementiert die Klasse `MatProcessor` ihre eigene Funktion `findContours`, welche die gleichnamige OpenCV-Funktion mit vordefinierten Parametern aufruft. Zusätzlich werden sehr kleine Konturen gefiltert, um das Rauschen in der Liste von gefundenen Konturen zu unterdrücken. Ist die Fläche einer Kontur kleiner als der Wert des Parameters `MIN_CONTOUR_AREA`, so wird sie aus der Ergebnismenge entfernt.

Die gezielte Suche nach Kreiskonturen wird mit der Funktion `findCircleContours` durchgeführt. Der einschließende Rahmen eines Kreises ist ein Quadrat und besitzt somit ein Seitenverhältnis von eins. Die Funktion `findCircleContours` berechnet das Seitenverhältnis für den Rahmen jeder gefundenen Kontur. Die Abweichung vom perfekten Seitenverhältnis wird verwendet, um zu bestimmen, ob es sich bei der Kontur um einen Kreis handelt. Ist die Abweichung kleiner oder gleich dem vom Parameter `CIRCLE_ASPECT_THRESHOLD` festgelegten Wert, so wird die Kontur als Kreiskontur behalten. Der Standardwert von 0,1 ergibt sich aus einer Reihe von Tests um festzustellen, ab welcher Abweichung eine Kreiskontur im Kamerabild nicht mehr annähernd als Kreis, sondern als Ellipse erkennbar ist.

Weiterhin stellt OpenCV die Funktion `fitEllipse` zur Verfügung. Sie wird verwendet, um eine passende Ellipse um eine Kontur zu berechnen. Als Parameter erwartet sie eine Kontur mit mindestens fünf Punkten. Die Funktion `findContoursForEllipseFit` in der Klasse `MatProcessor` filtert Konturen mit weniger als fünf Punkten heraus.

3.1.5 Die Klasse SampleAccumulator

Nunmehr können Marker hinsichtlich ihrer Maße und Position im Bild ausgewertet werden. Aufgrund des ständigen Verarbeitens neuer Kamerabilder macht es Sinn, gemessene Eigenschaften einer Kontur während einer Messung zu Sammeln um statistische Ausreißer zu unterdrücken. Die Klasse `SampleAccumulator` realisiert einen Messwertsammler, der Zahlenwerte aufnimmt und zum Ende einer Messung das arithmetische Mittel über alle aufgenommenen Werte bildet.

Intern wird bei der Instanziierung der Klasse ein `double`-Array angelegt mit einer Länge, die im Konstruktor festgelegt ist. Standardmäßig ist diese Länge auf 50 gesetzt, allerdings kann sie auch angepasst werden, um mehr oder weniger Messwerte aufzunehmen. Weiterhin wird ein `int` angelegt, der auf den Index im Array zeigt an dem der nächste übergebene Wert geschrieben wird. Werte können mit der Funktion `push` an den Messwertsammler übergeben werden. Der Rückgabewert der Funktion ist ein `boolean` und sagt aus, ob nach dem Hinzufügen des Messwertes noch weitere Werte aufgenommen werden können. Alternativ kann mit der Funktion `isFull` herausgefunden werden, ob alle angeforderten Messwerte gesammelt wurden.

Hat der Messwertsammler seine Maximalkapazität erreicht, so werden keine neuen Messwerte geschrieben. Durch den Aufruf der Funktion `clear` wird der Zeiger auf den Anfang des Arrays zurückgesetzt und alle Werte mit null überschrieben. Die Funktion `getAverage` berechnet das arithmetische Mittel über alle bisher aufgenommenen Messwerte. Somit ist es einfach, eine Messreihe zu starten, auszuwerten und gegebenenfalls zurückzusetzen.

3.1.6 Persistentes Speichern von Gleitkommazahlen

Android stellt mit der Klasse `SharedPreferences` einen persistenten Schlüssel-Werte-Speicher zur Verfügung [26]. Die Klasse ermöglicht die Speicherung von Anwendungsdaten und unterstützt die wichtigsten primitiven Datentypen, darunter `boolean`, `float`, `int` und `long`. Die Klasse unterstützt allerdings nicht den Datentyp `double`.

Um Messungen so genau wie möglich durchführen zu können, werden bei Berechnungen alle Teil- und Gesamtergebnisse als `double` gespeichert. Eine Typumwandlung von `double` zu `float` zum Zweck der Speicherung wäre möglich, würde aber mit einem großen Präzisionsverlust einhergehen.

Aus diesem Grund implementiert die Klasse `MainActivity` zwei Funktionen, welche der Syntax der Schreibe- und Lesefunktionen der Klasse `SharedPreferences` sehr ähneln: `putDouble` und `getDouble`. Die Funktion `putDouble` speichert eine Gleitkommazahl, indem sie mit der Funktion `doubleToRawLongBits` der `Double`-Klasse in ihre

Bitrepräsentation umgewandelt wird [27]. Der Rückgabewert der Funktion ist ein `long`, dessen Speicherung von der Klasse `SharedPreferences` unterstützt wird. Analog wird beim Aufruf der Funktion `getDouble` die Umkehroperation `longBitsToDouble` verwendet, um die Bitrepräsentation in einen `double` zu konvertieren. Somit ist es möglich, Gleitkommazahlen ohne Präzisionsverlust zu speichern.

3.2 Implementierung der Sichtfeldkalibrierung

Das Sichtfeld der Kamera wird für die stereoskopische Vermessung benötigt. Zwar stellt Android in den Kameraparametern einen horizontalen und vertikalen Sichtwinkel bereit, jedoch sind die Werte nicht verlässlich und berechnen Faktoren wie den Gerätezoom und das Seitenverhältnis der Kameravorschau nicht ein [8]. Aus diesem Grund erfolgt die Kalibrierung des Kamerasichtfeldes in der Anwendung.

Der Nutzer gibt in der Eingabemaske für die Sichtfeldkalibrierung den Radius des Markers und die Entfernung zum Marker ein. Diese Werte werden beim Start der Kalibrierung an die Klasse `FovActivity` übergeben. Sie wird mit einem `MatProcessor` und einem `SampleAccumulator` für den Radius des Markers initialisiert. Daraufhin wird dem Nutzer die Kameravorschau, wie in Abbildung 3.4 dargestellt, präsentiert.

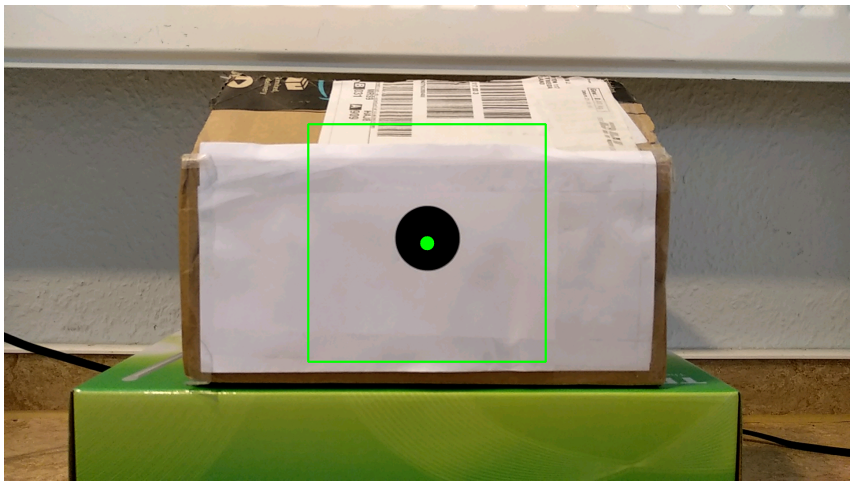

Abbildung 3.4: Oberfläche der Klasse FovActivity. Grün hervorgehoben sind der Markerbereich und das Zentrum des Bildschirms.

Da die Sichtfeldkalibrierung nur einmal durchgeführt wird, werden hohe Anforderungen an ihre Genauigkeit gestellt. Es wird eine quadratische Region mit einer festen Seitenlänge, die der Hälfte der Bildschirmhöhe entspricht, vorgegeben. Der Marker muss sich in diesem vorgegebenen Bereich befinden. Weiterhin wird ein Punkt in der Mitte des Bildschirms eingezeichnet. Der Radius dieses Punktes ist auf 1,5 % der Bildschirmhöhe festgelegt. Das Zentrum des Markers muss sich innerhalb des vorgegebenen Punktes befinden. Angewendet auf ein Gerät mit einer Bildschirmauflösung von 1920×1080 Pixel besitzt die quadratische Region eine Seitenlänge von 540 px und der Punkt in der

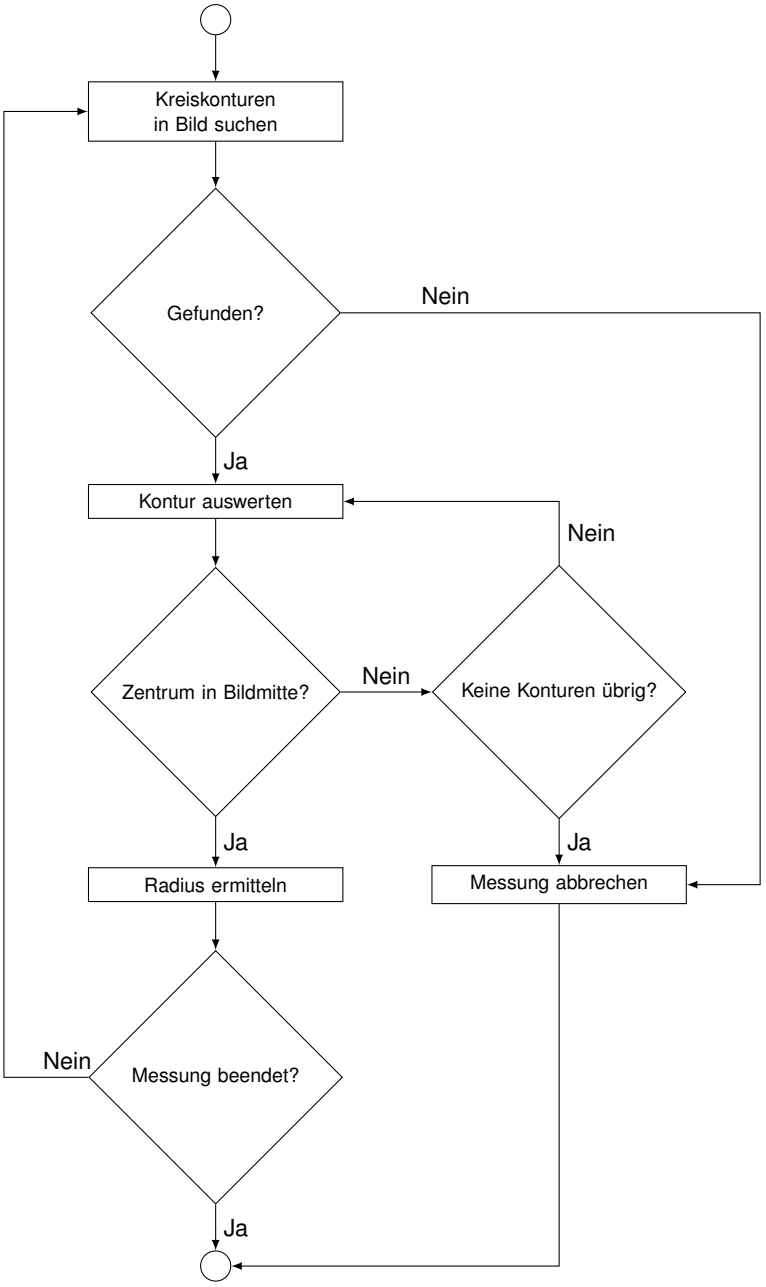

Abbildung 3.5: Ablaufdiagramm der Kontursuche in der Klasse FovActivity

Bildmitte einen Radius von 16,2 px. Die Nachkommastelle wird abgetrennt, weswegen der Radius tatsächlich 16 px beträgt.

Berührt der Nutzer den Bildschirm, so wird in der vorgegebenen Region nach Kreiskonturen gesucht. Findet der `MatProcessor` mindestens eine kreisförmige Kontur, so wird das Zentrum der Kontur lokalisiert. Liegt das Zentrum im Punkt der Bildmitte, so wird die Kontur als der gefundene Marker angenommen. Durch den Aufruf der OpenCV-Funktion `fitEllipse` wird eine Ellipse um die Kontur berechnet. Um den Radius des Markers zu erhalten, wird die Länge der Hauptachse der Ellipse halbiert. Der berechnete Radius wird dem `SampleAccumulator` hinzugefügt. Dieser Ablauf wird in Abbildung 3.5 dargestellt.

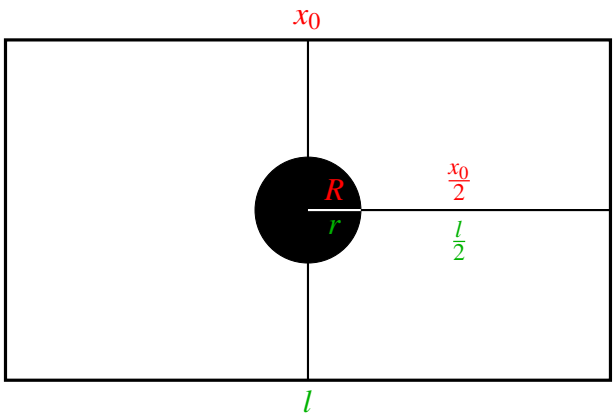

Abbildung 3.6: Verhältnisse von Pixel- und Realmaßen in der Sichtfeldkalibrierung. Rot sind Maße im Kamerabild. Grün sind Realmaße.

Ist der `SampleAccumulator` voll, also wurde der Radius des Markers 50 Mal bestimmt, so wird das Sichtfeld der Kamera berechnet. Die gemessenen Radien werden gemittelt und ergeben den Markerradius R in Pixeln. Dieser wird verwendet, um die Länge des sichtbaren Bereichs der Gegenstandsebene l zu berechnen. Da der tatsächliche Radius des Markers r als auch die Bildbreite x_0 in Pixel bekannt ist, ist es möglich l mit den in Abbildung 3.6 dargestellten Verhältnissen zu berechnen.

$$l = \frac{x_0}{R} r \tag{3.1}$$

Die mit Gleichung 3.1 berechnete Länge wird zusammen mit der vom Nutzer übergebenen Entfernung zum Marker verwendet, um das horizontale Sichtfeld der Kamera wie im Abschnitt zur Berechnung des Sichtfeldes zu ermitteln. Der berechnete Sichtwinkel wird als Rückgabewert der `FovActivity` gesetzt. Nach Beendigung der Activity wird dieser Wert vom zugehörigen Fragment ausgelesen und in den `SharedPreferences` der Anwendung gespeichert.

3.3 Implementierung des stereoskopischen Verfahrens

Der originale Messaufbau der stereoskopischen Vermessung nach Mrovlje und Vrančić sieht zwei identische, parallel zueinander aufgestellte Kameras vor, um die Entfernung zu beliebigen Markern zu berechnen. Mithilfe der in mobilen Endgeräten verbauten Ausrichtungssensoren ist es möglich, die Parallelität zwischen zwei Aufnahmeorten mit nur einer Kamera zu bewahren. In den folgenden Absätzen werden die Herausforderungen im Umgang mit der Android Sensor API beschrieben. Unter anderem wird ein Ansatz vorgestellt, welcher auf Grundlage der Geräteausrichtung die Messergebnisse des stereoskopischen Verfahrens verbessern kann.

3.3.1 Kontinuierlicher Ausrichtungswinkel

Der Drehvektorsensor wird verwendet, um die Lage des Gerätes im Raum zu beschreiben. Nach der Umwandlung der gemeldeten Werte des Sensors in eulersche Winkel können Roll-, Nick- und Gierwinkel des Gerätes ausgewertet werden. Der Gierwinkel besitzt jedoch einen begrenzten Wertebereich $\phi \in [-\pi, \pi)$. In diesem Abschnitt wird eine Möglichkeit für die Bestimmung der Geräteausrichtung über den Sensorwertebereich hinaus beschrieben.

Detektion der Wertebereichüberschreitung

Der begrenzte Wertebereich des Drehvektorsensors hat zur Folge, dass für Winkel nahe den Grenzen des Wertebereichs der Gierwinkel vom Positiven ins Negative und umgekehrt ausschlagen kann, obwohl sich die Ausrichtung des Gerätes nur minimal geändert hat.

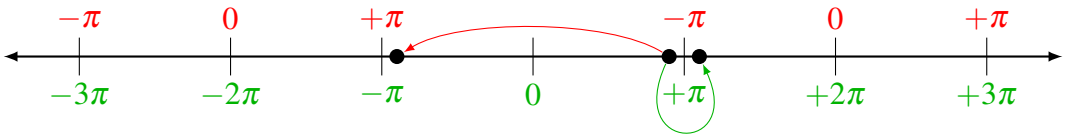

Abbildung 3.7: Veränderung des Gierwinkels des Drehvektorsensors beim Übertreten des Wertebereichs (rot) im Vergleich zur kontinuierlichen Winkelmessung (grün)

$$\Phi_t = 2\pi c_t + \phi_t \tag{3.2}$$

Das in Abbildung 3.7 dargestellte Problem kann gelöst werden, indem die Anwendung die Anzahl der Umdrehungen um die Ausrichtungsachse mitzählt. Sei ϕ_t der vom Sensor gemeldete Messwert zum jetzigen Zeitpunkt und c_t die Anzahl Umdrehungen um die entsprechende Achse. Dann kann der Gesamtwinkel Φ_t zum Zeitpunkt t mit der Gleichung 3.2 berechnet werden.

Die Zeit zwischen Messwerten kann vom Entwickler bestimmt werden, unterschreitet aber nie 200 ms [14]. Das bedeutet, dass zwischen den Messungen des Gierwinkels keine starken Änderungen auftreten sollten. Die einzige Ausnahme ist das Überschreiten des Wertebereichs. Die Änderung zwischen zwei zeitlich aufeinanderfolgenden Winkeln $\Delta\phi$, wie in Gleichung 3.3 dargestellt, kann also verwendet werden, um diese Überschreitung zu detektieren.

$$\Delta\phi = \phi_t - \phi_{t-1} \tag{3.3}$$

Wie erwähnt wird $\Delta\phi$, außer beim Übertreten des Wertebereichs, sehr klein sein. Durch Vergleichen von $\Delta\phi$ mit einem Schwellwert α, ab dem die Änderung des Winkels keiner natürlichen Bewegung entstammt, kann die Anzahl Umdrehungen c_t angepasst werden.

$$c_t = \left\{ \begin{array}{ll} c_{t-1}+1, & \text{wenn } \Delta\phi < -\alpha \\ c_{t-1}-1, & \text{wenn } \Delta\phi > \alpha \\ c_{t-1} & \text{sonst} \end{array} \right\} \tag{3.4}$$

Ist ϕ_t negativ und ϕ_{t-1} positiv, so ist die Differenz der beiden Winkel negativ. Unter der Annahme, dass der Winkel an keinen abgegrenzten Wertebereich gebunden ist, wäre ϕ_t weiterhin positiv. Somit muss eine Umdrehung aufaddiert werden. Analog gilt dies für den Wechsel vom Negativen ins Positive, nur dass eine Umdrehung subtrahiert werden muss. Diese Fallunterscheidung ist in Gleichung 3.4 zu sehen. Somit ist es möglich den Gierwinkel des Gerätes auch über den Wertebereich des Sensors hinaus zu berechnen.

Die Klasse ContinuousEulerHelper

Die Hilfsklasse `ContinuousEulerHelper` implementiert die Detektion der Wertebereichüberschreitung des Drehvektorsensors und berechnet aufgrund dessen die Ausrichtung des Gerätes. Sie stellt die Funktion `update` bereit, welche als Parameter den letzten gemeldeten Gierwinkel des Drehvektorsensors erwartet und verarbeitet. Zusätzlich kann mit der Funktion `getFullAngle` die aktuelle Gesamtausrichtung Φ_t des Gerätes abgefragt werden.

Die Klasse merkt sich stets den letzten gemessenen Gierwinkel um auf dessen Grundlage die Abweichung zum Winkel, welcher mit der Funktion `update` an die Klasse übergeben wird, auszuwerten. Diese Winkel entsprechen jeweils ϕ_{t-1} und ϕ_t in den obigen Ausführungen. Die Auswertung der Differenz zwischen den beiden Winkeln ist in Listing 3.2 dargestellt.

```
76 // Hat sich das Vorzeichen zwischen der letzten und der jetzigen Messung geändert?
77 if (Math.signum(this.mLastValue) != Math.signum(newValue)) {
78     float delta = newValue - this.mLastValue;
79
80     // Ist die Differenz der beiden Messwerte hinreichend groß?
81     if (Math.abs(delta) >= REVOLUTION_THRESHOLD) {
82         // Wenn das Vorzeichen der Änderung negativ ist, dann würde der gesamte
               Winkel
83         // weiter ins positive gehen. Das gleiche gilt andersrum. Deswegen muss das
84         // Zeichen umgekehrt werden um auf die Anzahl Umdrehungen schließen zu kö
               nnen.
85         this.mRevolutions -= (int) Math.signum(delta);
86         Log.d(TAG, "Sign changed. Revolutions: " + this.mRevolutions);
87     }
88 }
89
90 this.mLastValue = newValue;
```

Listing 3.2: Detektion der Wertebereichüberschreitung in der Klasse ContinuousEulerHelper

Wird der Wertebereich des Drehvektorsensors überschritten, so müssen die Vorzeichen von ϕ_{t-1} und ϕ_t verschieden sein. Ist diese Bedingung gegeben, so wird die Differenz der beiden Winkel mit dem Grenzwert α verglichen. Dieser Grenzwert ist in der Anwendung auf 1 rad, beziehungsweise rund 57,3°, festgelegt. Ist der Grenzwert überschritten, so wird der Zähler für die Umdrehungen des Gerätes um die Ausrichtungsachse angepasst. Zuletzt wird ϕ_t in Vorbereitung auf den nächsten gemeldeten Gierwinkel als ϕ_{t-1} definiert. Somit ist es möglich, die Geräteausrichtung auch an den Grenzen des Sensorwertebereiches verlässlich zu bestimmen.

3.3.2 Korrektur abweichender Kameraausrichtungen

Mit der Bestimmung der Ausrichtung der Gerätekamera ist es nicht nur möglich, die Parallelität zwischen den beiden Aufnahmeorten im stereoskopischen Verfahren zu bewahren. Kleine Abweichungen in den beiden Ausrichtungswinkeln können zudem korrigiert werden. In der Praxis treten minimale Abweichungen in den Ausrichtungen der Kamera zwischen den beiden Aufnahmen auf. Da das Bild auf eine bestimmte Anzahl von Pixeln begrenzt ist, kann ein kleiner Unterschied bereits einen großen Fehler bei der Berechnung der Distanz verursachen.

In Abbildung 3.8 wird angenommen, dass zuerst der Marker in der rechten und danach in der linken Bildhälfte vermessen wird. Somit wird die Position des Markers in der linken Bildhälfte aufgrund der Abweichung zum Ausrichtungswinkel, der während der Aufnahme des Markers in der rechten Bildhälfte ermittelt wurde, korrigiert. Die Ausrichtungswinkel der Kamera zwischen den beiden Aufnahmen unterscheiden sich um den Winkel ε. Es wird unterschieden, ob sich die Ausrichtung im mathematisch positiven oder negativen Drehsinn geändert hat.

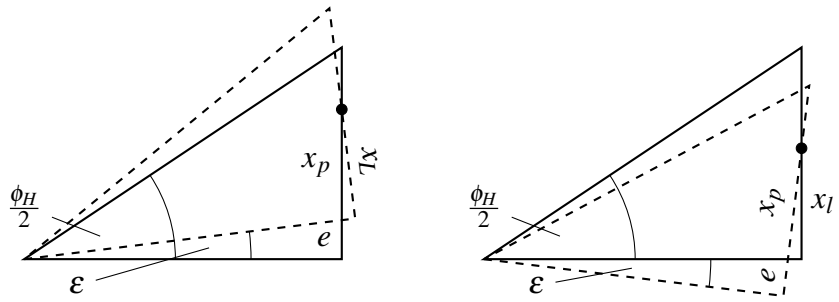

Abbildung 3.8: Korrektur abweichender Sichtwinkel zwischen Aufnahmen in der Stereoskopie. Links: Abweichung im mathematisch positiven Drehsinn. Rechts: Abweichung im mathematisch negativen Drehsinn.

Änderung im mathematisch positiven Drehsinn

Die tatsächliche Abweichung von der Bildmitte x_l setzt sich zusammen aus der von Winkel ε verstrichenen Strecke e und der gemessenen Abweichung x_L, welche auf die Objektebene projiziert werden muss. Die projizierte Strecke wird als x_p bezeichnet.

$$x_p = \frac{x_L}{\cos \varepsilon} \tag{3.5}$$

Die Bildebene steht senkrecht zum Betrachter und bildet mit der Ausrichtungsachse, auf der auch die Distanz zum Marker berechnet wird, ein rechtwinkliges Dreieck. Dementsprechend bilden auch x_L und x_p mit der Ausrichtungsachse der Kamera ein solches Dreieck. Somit kann x_p wie in Gleichung 3.5 berechnet werden.

$$e = \tan \varepsilon \frac{x_0}{\tan\left(\frac{\phi_H}{2}\right)} \tag{3.6}$$

Das Verhältnis von Sichtwinkel und Länge in der Projektionsebene unter der gleichen Ausrichtungsachse ist für alle Winkel gleich. Da die Breite des Kamerabildes x_0 und der horizontale Sichtwinkel der Kamera ϕ_H bekannt sind, kann die durch den Winkel ε verstrichene Strecke e wie in Gleichung 3.6 ermittelt werden.

Änderung im mathematisch negativen Drehsinn

Die gemessene Abweichung von der Bildmitte x_L setzt sich aus der Strecke e und der Strecke x_p zusammen, welche der Projektion der tatsächlichen Abweichung x_l auf die Bildebene entstammt. Die Strecke x_p kann ermittelt werden, indem e, dessen Bestimmung wie im vorhergehenden Abschnitt entsprechend der Gleichung 3.6 erfolgt, von der gemessenen Abweichung x_L abgezogen wird.

$$x_l = x_p \cos \varepsilon \tag{3.7}$$

Die tatsächliche Abweichung von der Bildmitte x_l und die projizierte Abweichung x_p bilden ein rechtwinkliges Dreieck. Somit kann x_l wie in Gleichung 3.7 ermittelt werden. Mit der Kenntnis wie sich Marker im Kamerabild bei Abweichungen im Ausrichtungswinkel zwischen Aufnahmen verschieben ist es nunmehr möglich, den daraus entstehenden Fehler zu korrigieren.

3.3.3 Die Klasse StereoscopyActivity

Der Nutzer gibt in der Eingabemaske für die stereoskopische Vermessung den Abstand zwischen den beiden Aufnahmeorten ein. Zusammen mit dem kalibrierten horizontalen Sichtwinkel wird die Eingabe an die Klasse `StereoscopyActivity` übergeben. Beim Start der Activity wird dem Nutzer die Kameravorschau mit einer vertikalen, grünen Linie durch die Mitte des Kamerabildes präsentiert. Die Activity arbeitet in zwei Phasen. In der ersten Phase wird der Marker in der rechten, in der zweiten Phase in der linken Bildhälfte vermessen. Der Ablauf der Messung wird in den folgenden Absätzen beschrieben. Das Schema der Markervermessung ist vereinfacht in Abbildung 3.9 dargestellt.

Drehvektorsensor und erste Phase

Beim Start der Activity werden ein `MatProcessor` und vier `SampleAccumulator` instanziiert: zwei für die horizontale Position des Markerzentrums in der rechten und linken Bildhälfte und zwei für die Geräteausrichtung während der Markervermessung in den jeweiligen Bildhälften. Zusätzlich wird ein `ContinuousEulerHelper` instanziiert. Die Activity registriert sich daraufhin als Listener für den Drehvektorsensor. Somit erhält sie in Intervallen von 60 ms die gemessene Ausrichtung des Gerätes als Quaternion. Während des gesamten Lebenszyklus der Activity wird der Gierwinkel des Gerätes aus die vom Drehvektorsensor bereitgestellte Quaternion berechnet und an den `ContinuousEulerHelper` übergeben, sodass jederzeit die Ausrichtung des Gerätes bestimmt werden kann.

In der ersten Phase wird der Marker in der rechten Bildhälfte vermessen. Nachdem der Nutzer die Kamera dementsprechend aufgestellt hat, zeichnet er die Region, in der sich der Marker befindet, ein. Im eingezeichneten Bereich wird daraufhin nach einer kreisförmigen Kontur gesucht. Wurde eine Kontur gefunden und wurde bestätigt, dass sie sich in der rechten Bildhälfte befindet, so wird die x-Koordinate des Markerzentrums im dafür zuständigen Messwertsammler gespeichert. Zeitgleich wird der Ausrichtungswinkel der Kamera aufgenommen und gespeichert. Erst wenn beide Messwertsammler 50 Werte besitzen, wird die erste Phase abgeschlossen und die zweite Phase initiiert.

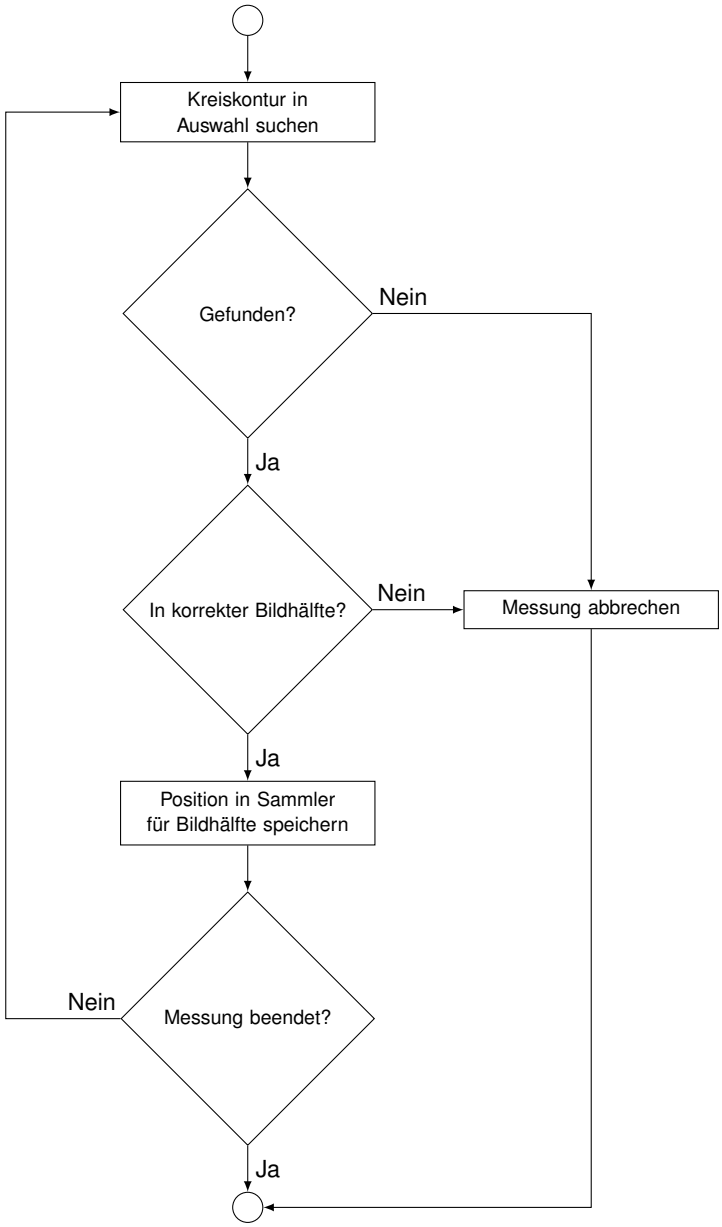

Abbildung 3.9: Ablaufdiagramm der Kontursuche in der Klasse StereoscopyActivity

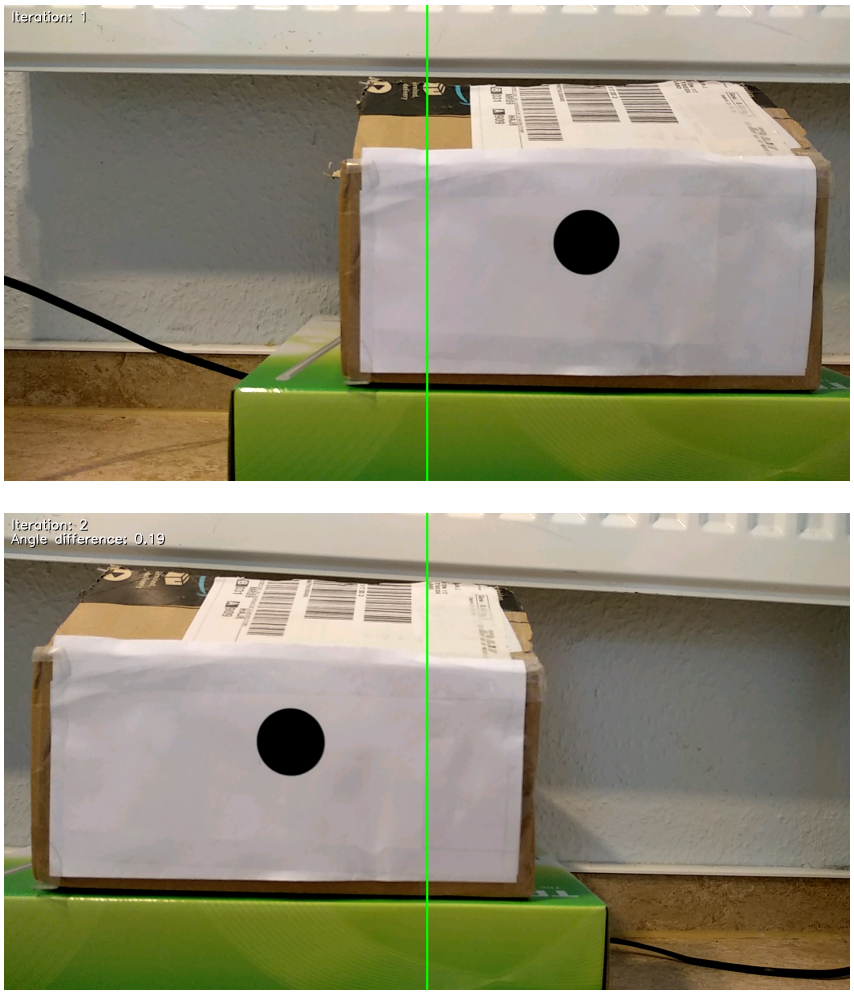

Abbildung 3.10: Benutzeroberfläche der Klasse StereoscopyActivity. Oben: erste Phase. Unten: zweite Phase.

Zweite Phase und Entfernungsberechnung

In der zweiten Phase wird der Marker in der linken Bildhälfte vermessen. Der Nutzer bewegt dabei die Kamera parallel zur Gegenstandsebene um die Strecke, die er vor dem Start der Messung angegeben hat. Neben der aktuellen Phase wird in der linken oberen Ecke des Kamerabildes, wie in Abbildung 3.10 zu sehen ist, die Abweichung der aktuellen Ausrichtung ϕ_l zum Mittel der gemessenen Ausrichtungswinkel während der ersten Phase $\overline{\phi_r}$ angezeigt. Um die Parallelität zwischen den beiden Aufnahmeorten zu gewährleisten, darf vor der Vermessung des Markers die Differenz der beiden Winkel $\phi_l - \overline{\phi_r}$ nicht größer als $\pm 1^\circ$ sein.

Der Nutzer zeichnet wieder die Region, in der sich der Marker befindet, ein. Nachdem überprüft wurde, ob sich der Marker in der linken Bildhälfte befindet, werden die x-Koordinate des Markerzentrums und der Ausrichtungswinkel der Kamera aufgenommen. Die Phase wird abgeschlossen, nachdem für die Markerposition als auch die Geräteausrichtung 50 Messwerte aufgenommen wurden.

```
292 private double calculateDistance(boolean applyError) {
293     double fovHalf = this.mHorizontalFov / 2d;
294     double fovHalfTan = Math.tan(fovHalf); // Es wird sowieso ausschließlich der
            Tangens verwendet.
295
296     // Horizontalen Versatz berechnen.
297     double xr = this.mStartX.getAverage() - this.mPreviewWidthHalf;
298     double xl = this.mStopX.getAverage() - this.mPreviewWidthHalf;
299
300     Log.d(TAG, "xr=" + xr + ", xl=" + xl);
301
302     if (applyError) {
303         // Verursachten Fehler durch Winkelabweichung berechnen.
304         double errorAngle = this.mStopYaw.getAverage() - this.mStartYaw.getAverage
                ();
305         double xError = Math.tan(Math.abs(errorAngle)) * this.mPreviewWidthHalf /
                fovHalfTan;
306
307         // Winkelabweichung auf Bildebene projizieren/anpassen.
308         if (Math.signum(errorAngle) == -1) {
309             // Kamera wurde im positiven Drehsinn gedreht.
310             double xProjected = xl / Math.cos(errorAngle);
311             xl = xProjected - xError;
312         } else {
313             // Kamera wurde im negativen Drehsinn gedreht.
314             double xProjected = xl + xError;
315             xl = Math.cos(errorAngle) * xProjected;
316         }
317
318         Log.d(TAG, "error angle=" + Math.toDegrees(errorAngle) + ", ex=" + xError);
319         Log.d(TAG, "corrected xl=" + xl);
320     }
321
322     return this.mCameraDistance * this.getPreviewSize().width / (2 * fovHalfTan * (
            xr - xl));
323 }
```

Listing 3.3: Berechnung der Entfernung zum Marker in der Klasse StereoscopyActivity

Die in Listing 3.3 aufgeführte Funktion `calculateDistance` berechnet nach dem Abschluss der beiden Phasen die Entfernung zum Marker. Sie besitzt einen booleschen Parameter, der beeinflusst ob die oben beschriebene Fehlerkorrektur angewendet wird oder nicht. Die Funktion wird zweimal aufgerufen. Beim ersten Aufruf wird die Entfernung zum Marker nach dem originalen Ansatz im Abschnitt zur Vermessung mit stereoskopischen Bildern berechnet.

Beim zweiten Aufruf wird die Fehlerkorrektur angewandt. Als Grundlage wird die Differenz der aufgenommenen Ausrichtungswinkel zwischen den beiden Aufnahmen berechnet. Anhand des Vorzeichens dieser Winkeldifferenz wird bestimmt, ob sich die Ausrichtung der Kamera im mathematisch positiven oder negativen Drehsinn geändert hat. Dementsprechend wird die Position des Markers in der linken Bildhälfte angepasst. Die Berechnung der Entfernung zum Marker erfolgt daraufhin analog zur Entfernungsberechnung ohne Fehlerkorrektur. Die beiden berechneten Entfernungen werden als Rückgabewert der `StereoscopyActivity` gesetzt und nach der Beendigung der Activity vom zugehörigen Fragment ausgelesen und angezeigt.

3.4 Implementierung des Verfahrens nach Cao et al.

Das Messverfahren nach Cao et al. basiert auf dem Lochkameramodell und wurde in der originalen Publikation an einer echten Lochkamera nachvollzogen. Der Marker und dessen Projektion wurden in deren Versuchen in Zentimetern vermessen. Auf einem mobilen Endgerät ist die Projektion des Markers nur in Pixeln messbar. In den folgenden Absätzen wird erläutert, wie in einem zusätzlichen Kalibrierungsschritt Realmaße mit Pixeln korreliert werden. Weiterhin wird beschrieben, wie die dadurch zusätzlich gewonnenen Informationen verwendet werden, um letzten Endes die Entfernung zum Marker zu berechnen.

3.4.1 Verknüpfung von Realmaßen und Pixeln

Das Lochkameramodell kann auf Kameras, die in Smartphones verbaut sind, angewendet werden. Da die verbauten Kameras meist monokular sind, besitzen sie eine fixe Brennweite. Allerdings kann die Fläche des Markers nur in Pixeln gemessen werden. Da die Fläche des Markers a konstant bleibt, muss für die reale Markergröße eine äquivalente Fläche in Pixel existieren.

$$\lim_{A \to a} f \sqrt{\frac{a}{A}} = f = d \tag{3.8}$$

Wenn sich die Fläche des projizierten Markers A der tatsächlichen Fläche des Markers a annähert, dann entspricht nach Gleichung 3.8 die Gegenstandsweite d der Brennweite f. Wenn während einer zusätzlichen Kalibrierung der Wert von d bekannt ist, dann kann die ursprüngliche Gleichung zur Entfernungsberechnung nach der Fläche des Markers a umgestellt werden.

$$a = A \frac{d^2}{f^2} \tag{3.9}$$

Somit ist es möglich, die reale Markergröße in eine Fläche von Pixeln zu übertragen. Wenn a nach der Gleichung 3.9 bestimmt wird, so kann die Entfernung für alle beliebigen gemessenen Flächen A eines Markers berechnet werden.

3.4.2 Die Klasse CalibrationProfile

Bevor Marker mit dem Verfahren nach Cao et al. vermessen werden können, müssen sie kalibriert werden. Wie im obigen Abschnitt erwähnt, muss auf einer vordefinierten Entfernung die Größe eines Markers in Pixeln vermessen werden. Zudem ist nicht jeder

Marker für alle Entfernungen geeignet. So wird beispielsweise die Vermessung eines Markers mit einem Radius von 2 cm auf einer Entfernung von 50 m keine robusten Ergebnisse liefern, da der Marker im Kamerabild kaum erkennbar sein wird. Durch die Verwendung von Kalibrierungsprofilen werden diese Probleme gelöst.

Kalibrierungsprofile sind durch die Klasse `CalibrationProfile` in der Anwendung repräsentiert. Eine Instanz besitzt vier Felder:

- `name`, ein frei wählbarer Name für den Marker
- `radius`, der Radius des Markers in cm
- `distance`, die Entfernung in cm auf welche der Marker kalibriert wurde
- `pixelRadius`, der Radius des Markers in px

Die Felder `name`, `radius` und `distance` werden durch den Nutzer festgelegt. Der Wert des Feldes `pixelRadius` wird während der Kalibrierung des Markers festgelegt. Nach der Kalibrierung sind Realmaß und Pixel für den Marker fest korreliert.

Weiterhin erbt die Klasse Funktionen von der Schnittstelle `Parcelable`. Sie stellt das Android-Äquivalent zur Java-Klasse `Serializable` dar und dient der Interprozesskommunikation im Betriebssystem Android [28]. Zum Starten einer Activity können neben primitiven Datentypen, Arrays und einigen häufig verwendeten komplexen Datentypen in Java auch eigene Klassen als Parameter übergeben werden, sofern sie die Schnittstelle `Parcelable` erben. Der Aufbau der Klasse `CalibrationProfile` und die Möglichkeit ihrer Serialisierung und Deserialisierung sind wichtige Kenntnisse, um die Implementierung der Vermessung nach Cao et al. nachzuvollziehen.

3.4.3 Die Klasse CaoActivity

Die Vermessung nach Cao et al. ist in der Klasse `CaoActivity` implementiert. Sie arbeitet in zwei verschiedenen Modi: einem Kalibrierungsmodus und einem Messmodus. Beide vermessen den Radius des Markers in Pixeln, jedoch sind die Rückgabewerte verschieden. In den folgenden Absätzen werden die Gemeinsamkeiten und Unterschiede der beiden Modi dargestellt.

Kalibrierungsmodus

Der Nutzer gibt für die Kalibrierung eines neuen Markers dessen Radius, die Entfernung, auf die der Marker kalibriert werden soll, und einen Namen für den Marker in der Eingabemaske für die Vermessung nach Cao et al. ein. Vor dem Start der `CaoActivity` wird mit den Nutzereingaben eine Instanz von `CalibrationProfile` erstellt. Das Feld für den Markerradius in Pixeln bleibt dabei vorerst unbesetzt. Das Kalibrierungsprofil wird als Parameter zum Start der `CaoActivity` übergeben.

Beim Start der `CaoActivity` wird ein `MatProcessor` und ein `SampleAccumulator` für die gemessenen Markerradien in Pixel instanziiert. Die Brennweite der Kamera wird aus den Kameraparametern gelesen. Der Nutzer wird mit der Kameravorschau präsentiert und zeichnet die Region ein, in der sich der Marker befindet. Im ausgewählten Bereich wird mithilfe des `MatProcessor` nach kreisförmigen Konturen gesucht. Wurde eine Kontur gefunden, so wird diese als der aufgestellte Marker angenommen. Analog der Beschreibung im Abschnitt zur Implementierung der Sichtfeldkalibrierung wird der Radius als die Hälfte der Hauptachsenlänge der Ellipse um den Marker berechnet. Die Ellipse wird durch die OpenCV-Funktion `fitEllipse` gegeben.

Wurde der Markerradius 50 Mal gemessen, so wird die Messung beendet. Das übergebene `CalibrationProfile` wird um das Mittel der gemessenen Markerradien in Pixel ergänzt und als Rückgabewert der Activity gesetzt. Das nun vollständige Profil wird nach der Beendigung der Activity vom `CaoFragment` in den `SharedPreferences` der Anwendung gespeichert.

Messmodus

Die gespeicherten und kalibrierten Profile werden zum Start der Anwendung aus den `SharedPreferences` gelesen und in einer ausklappbaren Liste in der Eingabemaske für die Vermessung nach Cao et al. zusammengefasst. Die Auswahl ist in Abbildung 3.11 dargestellt. Vor dem Start einer Messung muss der Nutzer das Profil für den zu vermessenden Marker auswählen. Das vollständige Kalibrierungsprofil wird als Parameter beim Start der `CaoActivity` übergeben.

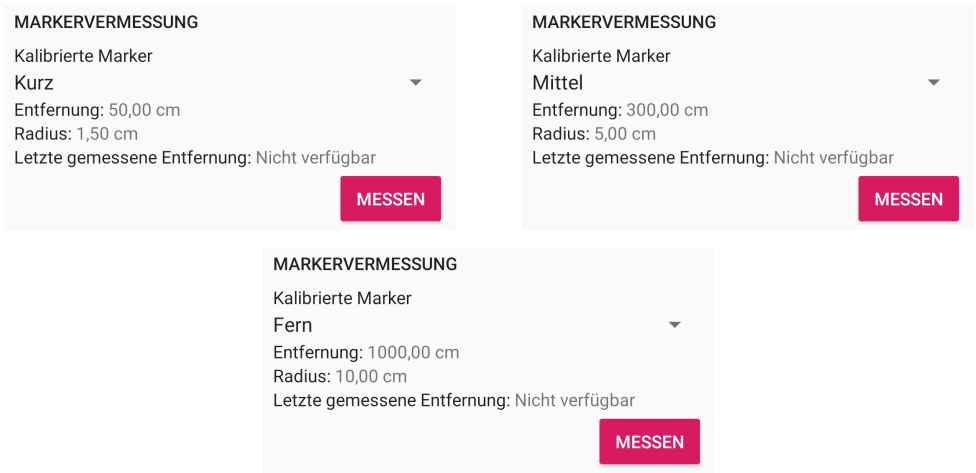

Abbildung 3.11: Auswahl der gespeicherten Marker in der Eingabemaske für die Vermessung nach Cao et al.

Die Suche nach dem Marker und dessen Vermessung verläuft analog zum Kalibrierungsmodus. Nach der Vermessung des Markers wird anhand des übergebenen Kalibrierungsprofils die theoretische Größe des Markers a im Brennpunkt der Kamera in

Pixel, wie oben beschrieben, berechnet. Aus dem Mittelwert der gemessenen Markerradien wird die Fläche des Kreismarkers A berechnet. Zusammen mit der Brennweite der Kamera f wird die Entfernung zum Marker wie im Abschnitt zur Vermessung nach Cao et al berechnet. Die ermittelte Distanz wird als Rückgabewert der `CaoActivity` gesetzt und vom `CaoFragment` nach Beendigung der Activity angezeigt.

4 Messversuche und Auswertung

Um die beiden Entfernungsmessverfahren zu vergleichen, werden drei Messreihen, wie in Tabelle 4.1 aufgeführt, aufgenommen. Sie testen die Genauigkeit der Verfahren auf kurze, mittlere und weite Distanzen. Jede Entfernung wird drei Mal gemessen. Das Testgerät ist ein Xiaomi Mi A1 mit der Android Version 9 und den letzten Sicherheitsupdates vom 1. August 2019. Die Auflösung der Kameravorschau beträgt 1920×1080 Pixel und die Brennweite der Kamera liegt bei 3,83 mm. Die Position des Gerätes wird mit einem Stativ fixiert.

Tabelle 4.1: Eigenschaften der Messreihen

Messreihe		**Kurz**	**Mittel**	**Lang**
Markerradius in cm		1,5	5	10
Entfernungen in cm	d_1	20	100	600
	d_2	35	200	800
	d_3	50	300	1000
	d_4	65	400	1200
	d_5	80	500	1400
Kameraabstand in cm[1]		20	80	400
Kalibrierungsentfernung in cm[2]		50	300	1000

Die Entfernung wird zu einem schwarzen Kreismarker auf einem weißen Hintergrund gemessen. Die Markerradien für die einzelnen Messreihen sind so ausgewählt, sodass sich die Größe der Marker im Bild zwischen den zu messenden Entfernungen sichtbar verändert. Das Testgerät wird parallel zum Marker aufgestellt. Nach einigen Tests mit Messungen im Innenbereich wurde festgestellt, dass der Drehvektorsensor aufgrund von Schwankungen im Magnetfeld keine verlässlichen Werte liefert. Die Messungen werden im Außenbereich durchgeführt, um diese Schwankungen zu minimieren. Der Messaufbau ist in Abbildung 4.2 dargestellt.

Die in Tabelle 4.1 aufgeführten Kameraabstände für die stereoskopische Vermessung wurden so bestimmt, sodass auf der kleinsten Entfernung in der Messreihe der Marker im Bild den größtmöglichen horizontalen Versatz aufweist. Für die Vermessung nach Cao et al. wurden die Marker auf der mittleren Entfernung der Messreihe kalibriert.

Die Sichtfeldkalibrierung wurde mit einem Marker mit einem Radius von 1,5 cm auf eine Entfernung von 20 cm durchgeführt. Das horizontale Sichtfeld der Gerätekamera wurde

[1] Betrifft nur die stereoskopische Vermessung.
[2] Betrifft nur die Vermessung nach Cao et al.

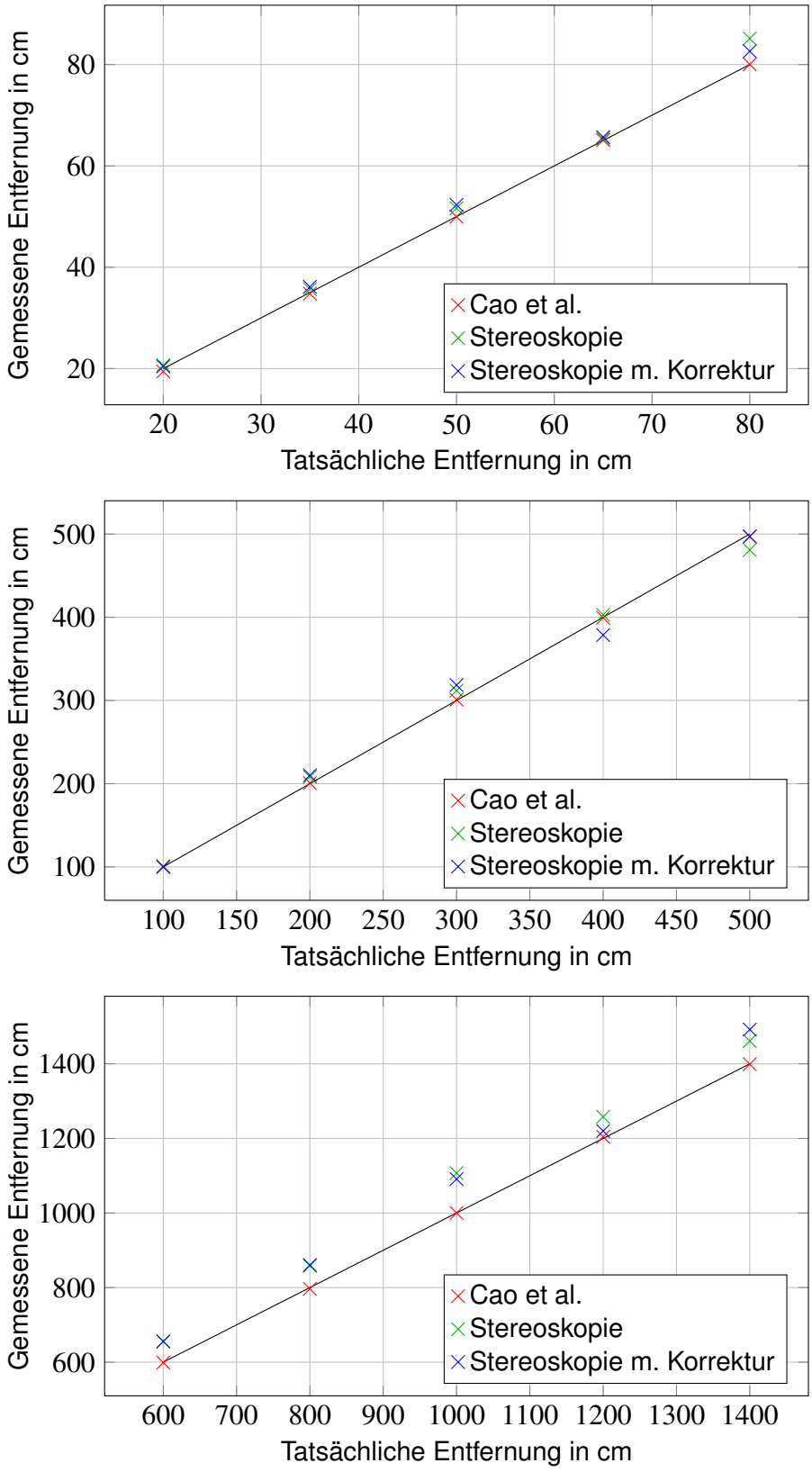

Abbildung 4.1: Vergleich der gemessenen Entfernungen zwischen den implementierten Messverfahren

mit 65,06° berechnet. Die Standardabweichung des Markerradius in Pixeln liegt bei 0,083 180 px.

Abbildung 4.2: Aufbau des Messversuchs

Die Graphen in Abbildung 4.1 stellen die gemessene Entfernung der implementierten Messverfahren in Abhängigkeit der tatsächlichen Entfernungen dar. Die Messwerttabellen, welche die Grundlage für die abgebildeten Graphen stellen, befinden sich in Anhang A. Sie zeigen deutlich, dass das Verfahren nach Cao et al. wesentlich robuster ist als die stereoskopische Vermessung. Bis auf eine Ausnahme bleibt der relative Fehler bei der Vermessung nach Cao et al. im Bereich von unter 1 %, häufiger sogar unterhalb von 0,5 %. Der vergleichsweise hohe Fehler bei der Messung auf 20 cm ist dem fixierten Fokus der Kamera verschuldet. Da der Fokus auf einer gedachten Ebene liegt, welche sich unendlich weit von der Kamera befindet, erscheinen Objekte, welche sich sehr nah an der Kamera befinden, etwas unscharf. Die verschwommenen Ränder des Markers vergrößern dessen Umriss und lassen somit den Marker größer erscheinen als er wirklich ist.

Der stereoskopische Ansatz zeigt für die Messungen auf kurze und mittlere Distanzen einen höheren relativen Fehler bis zu 6,42 %. Die Abweichung vom tatsächlichen Wert steigt bei der Messreihe für lange Distanzen auf bis zu 10,7 %. Eine bedeutsame Verbesserung durch die Fehlerkorrektur mithilfe der Vermessung der Geräteausrichtung kann nicht nachgewiesen werden. Der Hauptgrund für diese Ungenauigkeit ist der Drehvektorsensor.

$$\Delta d = \frac{d^2}{b} \tan(\Delta\phi) \tag{4.1}$$

Das Problem wird in Abbildung 4.3 für die Messungen auf 50 cm dargestellt. Die Streuung der x-Koordinate des Markers im Kamerabild befindet sich im Subpixelbereich und

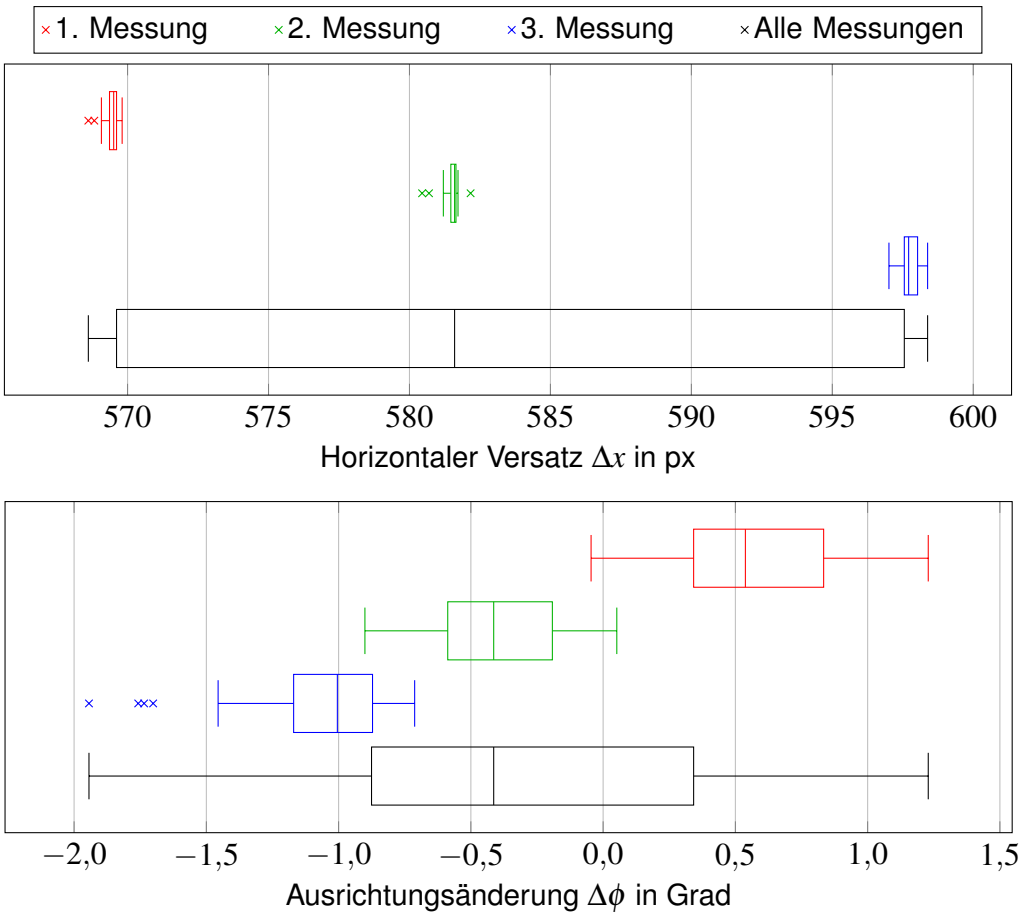

Abbildung 4.3: Messwertstreuung über alle Messungen im Vergleich zu einzelnen Messungen bei stereoskopischer Vermessung. Entfernung: 50 cm.

ist somit sehr gering. Allerdings beläuft sich die Streuung des vom Drehvektorsensors gemessenen Ausrichtungswinkels bei einzelnen Messungen auf etwa $\pm 0{,}5^\circ$ um den Mittelwert. Durch abweichende Kameraausrichtungen zwischen den beiden Aufnahmeorten entsteht ein Messfehler. In den Betrachtungen von Mrovlje und Vrančić [6] wird die Gleichung 4.1 vorgegeben, mit welcher der Messfehler berechnet werden kann.

Am Beispiel der Messung auf 50 cm mit einem Kameraabstand von 20 cm und einer Abweichung im Ausrichtungswinkel von 1° ergibt sich somit ein Fehler von 2,18 cm. Die Unterschiede zwischen gemessenen und tatsächlichen Entfernungen sind für die Messreihen auf kurze und mittlere Distanzen im verträglichen Bereich. Bei der Messreihe auf lange Distanzen sind lediglich die gemessenen Entfernungen auf 1200 cm und 1400 cm innerhalb des akzeptablen Fehlerbereichs. So dürfte beispielsweise auf 600 cm der Messfehler höchstens 15,71 cm betragen statt der gemessenen 55,06 cm.

Ein weiterer Grund ist der oben erwähnte Einfluss des Magnetfeldes auf das Testgerät und somit auch auf den Drehvektorsensor. Während der Messungen hat sich herausgestellt, dass auch im Außenbereich der Drehvektorsensor sehr anfällig gegenüber kleinen Änderungen im Magnetfeld ist. In der Nähe metallischer Gegenstände meldet der Sen-

sor teils sehr große Abweichungen im Bereich von etwa 10° trotz kleiner Änderungen an der tatsächlichen Ausrichtung des Gerätes. Somit ist die Wahrscheinlichkeit sehr hoch, dass die Kamera zwischen den beiden Aufnahmen nicht gleich ausgerichtet ist, obwohl sich die gemessene Abweichung im Toleranzbereich von $\pm 1°$ befindet. Weiterhin beeinträchtigt die Ungenauigkeit des Sensors die Ergebnisse der Fehlerkorrektur. Somit erzielt auch der stereoskopische Ansatz mit Fehlerkorrektur auf Grundlage der Ausrichtungsänderung der Kamera selten bessere Ergebnisse.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden zwei Entfernungsmessverfahren in einer Anwendung für das mobile Betriebssystem Android implementiert und hinsichtlich ihrer Genauigkeit miteinander verglichen. Dabei hat sich die Vermessung nach Cao et al. als wesentlich genauer und weniger fehleranfällig als die stereoskopische Vermessung erwiesen. Messfehler entstehen hauptsächlich durch die Verwendung des Drehvektorsensors zur Bestimmung der Geräteausrichtung. Die Ungenauigkeit des Sensors erlaubt keine Feststellung der Parallelität der Kamera zwischen den beiden Aufnahmeorten. Weiterhin muss das Sichtfeld der Gerätekamera vor Beginn der Messung kalibriert werden, da die Android API diesbezüglich keine verlässlichen Werte bereitstellt. Dementsprechend ist es schwer, die Genauigkeit der Sichtfeldkalibrierung zu bewerten.

Die Vermessung nach Cao et al. ist lediglich von der Brennweite der Kamera abhängig, welche von der Android API zur Verfügung gestellt wird. Da das Verfahren lediglich auf Bildbasis arbeitet und keiner weiteren Messwerte aus anderen Quellen bedarf, ist die Genauigkeit des Verfahrens lediglich von der Kalibrierung der zu vermessenden Marker abhängig. Somit können die in dieser Arbeit geprüften Entfernungen bis auf den Zentimeter genau vermessen werden.

Unabhängig vom Messverfahren können die Ergebnisse verbessert werden, indem statt eines fixierten Kamerafokus ein intelligenter Autofokus implementiert wird. Dieser kann den Softwarefokus des Gerätes stets auf den zu vermessenden Marker fokussieren und somit Messfehler durch unscharfe Abgrenzungen des Markers von seinem Hintergrund minimieren. Weiterhin kann die Bildvorverarbeitung und Kontursuche auf die Erkennung von Markern in variierenden Umweltbedingungen ausgeweitet werden. Somit können auch Marker außerhalb von idealen Umständen erkannt werden.

Zukünftige Forschung könnte sich der Anwendung des Verfahrens nach Cao et al. auf beliebige Marker widmen. Da die Entfernungsberechnung von der Fläche des Markers abhängig ist, können auch Marker vermessen werden, welche nicht strikt kreisförmig sind. Weiterhin könnte das Verfahren von Markern auf beliebige Konturen ausgeweitet werden. Beispielsweise können hierfür Konzepte aus dem maschinellen Lernen zur Objekterkennung verwendet werden, um ihre Maße zu schätzen. Unter anderem gilt es die Grenzen des Messverfahrens zu erforschen über die in dieser Arbeit vermessenen Entfernungen hinaus.

Anhang A: Messwerttabellen

Tabelle A.1: Messergebnisse mit dem Verfahren nach Cao et al.

d **in cm**	d_1 **in cm**	d_2 **in cm**	d_3 **in cm**	$\overline{d}$ **in cm**	$\overline{d}-d$ **in cm**	δd **in %**
20	19,40	19,40	19,47	19,423	–0,577	–2,88
35	34,75	34,82	34,79	34,787	–0,213	–0,61
50	49,98	49,97	49,91	49,953	–0,047	–0,09
65	65,08	65,06	65,08	65,073	0,073	0,11
80	80,09	80,11	79,92	80,040	0,040	0,05
100	100,59	100,73	100,66	100,660	0,660	0,66
200	200,99	200,24	200,97	200,733	0,733	0,37
300	301,65	301,07	300,06	300,927	0,927	0,31
400	399,90	399,76	398,65	399,437	–0,563	–0,14
500	498,00	497,14	496,03	497,057	–2,943	–0,59
600	599,78	599,37	597,73	598,960	–1,040	–0,17
800	797,77	796,71	795,42	796,633	–3,367	–0,42
1000	990,78	1002,47	1007,42	1000,223	0,223	0,02
1200	1208,67	1203,19	1200,77	1204,210	4,210	0,35
1400	1395,74	1402,76	1399,00	1399,167	–0,833	–0,06

Tabelle A.2: Messergebnisse mit der stereoskopischen Vermessung

d **in cm**	d_1 **in cm**	d_2 **in cm**	d_3 **in cm**	$\overline{d}$ **in cm**	$\overline{d}-d$ **in cm**	δd **in %**
20	20,72	20,47	20,85	20,680	0,680	3,40
35	36,06	36,11	34,83	35,667	0,667	1,90
50	52,86	51,76	50,36	51,660	1,660	3,32
65	66,67	61,01	68,44	65,373	0,373	0,57
80	83,14	84,46	87,81	85,137	5,137	6,42
100	100,05	100,66	100,83	100,513	0,513	0,51
200	206,64	206,48	211,30	208,140	8,140	4,07
300	311,80	313,61	310,22	311,877	11,877	3,96
400	416,70	400,84	391,25	402,930	2,930	0,73
500	508,76	449,11	484,91	480,927	–19,073	–3,81
600	657,88	651,88	658,96	656,173	56,173	9,36
800	840,25	878,75	855,69	858,230	58,230	7,28
1000	1120,97	1101,46	1098,58	1107,003	107,003	10,70
1200	1304,68	1203,29	1266,00	1257,990	57,990	4,83
1400	1468,79	1438,09	1476,74	1461,207	61,207	4,37

Tabelle A.3: Messergebnisse mit der korrigierten stereoskopischen Vermessung

d **in cm**	d_1 **in cm**	d_2 **in cm**	d_3 **in cm**	$\overline{d}$ **in cm**	$\overline{d}-d$ **in cm**	δd **in %**
20	20,37	20,34	20,58	20,430	0,430	2,15
35	36,56	36,57	35,36	36,130	1,130	3,23
50	51,40	52,72	52,84	52,320	2,320	4,64
65	65,29	64,73	67,04	65,687	0,687	1,06
80	82,62	82,34	82,92	82,627	2,627	3,28
100	99,36	99,88	100,38	99,873	–0,127	–0,13
200	207,99	209,27	213,14	210,133	10,133	5,07
300	327,33	328,54	299,61	318,493	18,493	6,16
400	397,13	375,61	363,21	378,650	–21,350	–5,34
500	498,00	497,14	496,03	497,057	–2,943	–0,59
600	667,11	656,54	641,53	655,060	55,060	9,18
800	856,42	866,54	858,51	860,490	60,490	7,56
1000	1122,32	1078,44	1071,91	1090,890	90,890	9,09
1200	1267,53	1124,48	1267,40	1219,803	19,803	1,65
1400	1541,07	1477,57	1457,07	1491,903	91,903	6,56

Literatur

[1] VentureBeat. *Prognose zur Anzahl der Smartphone-Nutzer weitweit von 2016 bis 2021 (in Milliarden)*. September 2018. URL: https://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/ (besucht am 27. 09. 2019).

[2] Apple Inc. *Using the Measure app on your iPhone, iPad or iPod touch*. Mai 2019. URL: https://support.apple.com/en-us/HT208924 (besucht am 27. 09. 2019).

[3] Google Developers. *Distribution dashboard*. Mai 2019. URL: https://developer.android.com/about/dashboards/ (besucht am 27. 09. 2019).

[4] Klaus Tönnies. *Grundlagen der Bildverarbeitung*. München Pearson Studium, 2005. ISBN: 3827371554.

[5] Samuel Gibbs. *Lytro Illum camera lets users refocus blurred photos after shooting*. April 2014. URL: https://www.theguardian.com/technology/2014/apr/22/lytro-illum-camera-refocus-blurred-photos (besucht am 01. 08. 2019).

[6] Jernej Mrovlje und Damir Vrancic. "Distance measuring based on stereoscopic pictures". In: *9th International PhD workshop on systems and control: young Generation Viewpoint*. Bd. 2. 2008, S. 1–6.

[7] Ashutosh Saxena, Sung H Chung und Andrew Y Ng. "Learning depth from single monocular images". In: *Advances in neural information processing systems*. 2006, S. 1161–1168.

[8] StackOverflow. *Determine angle of view of smartphone camera*. Juli 2010. URL: https://stackoverflow.com/a/3261794 (besucht am 17. 09. 2019).

[9] Yu-Tao Cao u. a. "Circle Marker Based Distance Measurement Using a Single Camera". In: *Lecture Notes on Software Engineering* 1.4 (2013), S. 376.

[10] Google Developers. *Activity*. September 2019. URL: https://developer.android.com/reference/android/app/Activity (besucht am 20. 09. 2019).

[11] Google Developers. *Understand the Activity Lifecycle*. August 2019. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (besucht am 15. 08. 2019).

[12] Google Developers. *Camera API*. August 2019. URL: https://developer.android.com/guide/topics/media/camera (besucht am 08. 08. 2019).

[13] Google Developers. *CameraX Architecture*. August 2019. URL: https://developer.android.com/training/camerax/architecture (besucht am 02. 08. 2019).

[14] Google Developers. *Sensors Overview*. August 2019. URL: https://developer.android.com/guide/topics/sensors/sensors_overview (besucht am 02. 08. 2019).

[15] Google Developers. *Sensor types*. August 2019. URL: https://source.android.com/devices/sensors/sensor-types (besucht am 24. 08. 2019).

[16] Google Developers. *SensorManager*. September 2019. URL: https://developer.android.com/reference/android/hardware/SensorManager.html (besucht am 19. 09. 2019).

[17] John Vince. *Geometric Algebra for Computer Graphics*. eng. London Springer, 2008. ISBN: 9781846289972.

[18] OpenCV Team. *About — OpenCV*. August 2019. URL: https://opencv.org/about/ (besucht am 08. 08. 2019).

[19] OpenCV Contributors. *OpenCv 3.4.1 for Android: Bad JavaCamera2View performance*. April 2018. URL: https://github.com/opencv/opencv/issues/11261 (besucht am 19. 09. 2019).

[20] OpenCV Contributors. *OpenCV Manager on Google Play Store needs update to 3.2.0*. Februar 2017. URL: https://github.com/opencv/opencv/issues/8120 (besucht am 19. 09. 2019).

[21] OpenCV Dev Team. *Android Development with OpenCV — OpenCV 2.4.13.7 documentation*. September 2019. URL: https://docs.opencv.org/2.4/doc/tutorials/introduction/android_binary_package/dev_with_OCV_on_Android.html (besucht am 19. 09. 2019).

[22] Google Developers. *Fragment*. September 2019. URL: https://developer.android.com/reference/android/app/Fragment.html (besucht am 20. 09. 2019).

[23] OpenCV Dev Team. *Image Filtering — OpenCV 2.4.13.7 documentation*. September 2019. URL: https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#bilateralfilter (besucht am 20. 09. 2019).

[24] OpenCV Dev Team. *Structural Analysis and Shape Descriptors — OpenCV 2.4.13.7 documentation*. September 2019. URL: https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#findcontours (besucht am 20. 09. 2019).

[25] Maximilian Jugl. *Konturerkennung und -verfolgung in Echtzeit mit Gerätekameras auf dem mobilen Betriebssystem Android*. Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften. Juni 2019.

[26] Google Developers. *Save key-value data*. September 2019. URL: https://developer.android.com/training/data-storage/shared-preferences (besucht am 20. 09. 2019).

[27] Oracle. *Double (Java Platform SE 7)*. 2018. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/Double.html (besucht am 28. 09. 2019).

[28] Jens Vogel. *Understanding Android Parcelable — Tutorial*. April 2017. URL: https://www.vogella.com/tutorials/AndroidParcelable/article.html (besucht am 16. 09. 2019).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 21. Oktober 2019